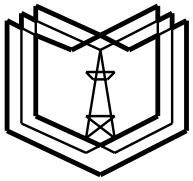

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное бюджетное образовательное учреждение высшего образования «КАЗАНСКИЙ ГОСУДАРСТВЕННЫЙ ЭНЕРГЕТИЧЕСКИЙ УНИВЕРСИТЕТ»

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Практикум

Казань
2018

УДК 681.3.068
ББК 32.973.26-018.1
О29

О29 Объектно-ориентированное программирование: практикум / сост. А. А. Халидов. – Казань: Казан. гос. энерг. ун-т, 2018. – 83 с.

Содержит описание пяти лабораторных работ по дисциплине «Объектно-ориентированное программирование». В каждой лабораторной работе приведены краткие теоретические сведения, сопровождаемые примерами, типовые варианты заданий и контрольные вопросы, позволяющие определить уровень знаний студентов. При выполнении лабораторных работ программы составляются в среде Microsoft Visual C#.

Предназначен для студентов очной формы обучения по образовательной программе «Математическое и программное обеспечение систем обработки информации и управления» направления подготовки 01.03.04 «Прикладная математика»; по образовательной программе «Программное обеспечение средств вычислительной техники и автоматизированных систем» направления подготовки 09.03.01 «Информатика и вычислительная техника».

УДК 681.3.068
ББК 32.973.26-018.1

ВВЕДЕНИЕ

Лабораторный практикум по дисциплине «Объектно-ориентированное программирование» (ООП) содержит пять работ, описание которых составлено по типовой схеме и включает:

– тему и цель лабораторной работы;

– краткие теоретические сведения с примерами решения задач, необходимые для выполнения лабораторной работы;

– порядок выполнения лабораторной работы;

– задачи для самостоятельного решения в ходе лабораторной работы, где номер варианта задачи соответствует порядковому номеру фамилии студента в журнале группы;

– список вопросов для самоконтроля.

При сдаче зачета по лабораторной работе студент должен представить письменный отчет, который содержит постановку задачи, исходные данные, алгоритм решения задачи, листинг программы, результат выполнения программы. Образец оформления отчета приведен в Приложении. Также студент должен устно ответить на вопросы преподавателя по теме лабораторной работы.

Лабораторные работы проводятся с целью формирования у студентов теоретических знаний и практических навыков разработки программ, решения прикладных задач на компьютере с использованием объектно-ориентированных технологий и направлены на формирование следующих компетенций:

– способность использовать современные математические методы и прикладные программные средства и осваивать современные технологии программирования;

– способность использовать стандартные пакеты прикладных программ для решения практических задач на электронных вычислительных машинах, отлаживать и тестировать прикладное программное обеспечение.

В результате освоения дисциплины студент должен:

– знать современные языки объектно-ориентированного программирования и системы программирования, существующие подходы к верификации моделей программного обеспечения;

– уметь разрабатывать прикладные программы на основе языков и систем объектно-ориентированного программирования;

– владеть основными приемами объектно-ориентрованного программирования.

Лабораторная работа №1

КЛАССЫ И ОБЪЕКТЫ, ИНКАПСУЛЯЦИЯ, НАСЛЕДОВАНИЕ

Цель работы: овладение навыками проектирования простейших классов и создания объектов класса; освоение принципов инкапсуляции и наследования.

Краткие теоретические сведения

Язык программирования высокого уровня C# является полноценным объектно-ориентированным языком. Это значит, что программу на C# можно представить в виде взаимосвязанных взаимодействующих между собой объектов.

Описание объекта – класс, а объект – экземпляр этого класса. Можно еще провести следующую аналогию. У всех есть некоторое представление о человеке – руки, ноги, голова, пищеварительная и нервная система, головной мозг и т. д. – некоторый шаблон. Этот шаблон можно назвать классом. Реально же существующий человек, фактически экземпляр данного класса, является объектом класса.

Класс определяется с помощью ключевого слова **class**:

```
class Book{}
```

Вся функциональность класса представлена его членами – полями (полями называются переменные класса), свойствами, методами, событиями. Для примера создадим класс **Book**, в котором будут храниться переменные – название, автор и год издания книги. Кроме того, класс будет содержать метод для вывода информации о книге на консоль:

```
class Book
{
    public string name;
    public string author;
    public int year;

    public void Info()
    {
        Console.WriteLine("Книга '{0}' (автор {1}) была
издана в {2} году", name, author, year);
         }
}
```

Кроме обычных методов в классах используются и специальные методы, которые называются конструкторами. Конструкторы вызываются при создании нового объекта данного класса. Отличительной чертой конструктора является то, что его название должно совпадать с названием класса:

```
class Book
{
    public string name;
    public string author;
    public int year;
      // конструктор без параметров
    public Book()
    { }
     // конструктор с параметрами
    public Book(string name, string author, int year)
    {
        this.name = name;
        this.author = author;
        this.year = year;
    }

    public void Info()
    {
        Console.WriteLine("Книга '{0}' (автор {1}) была из-
дана в {2} году", name, author, year);
    }
}
```

Одно из назначений конструктора – начальная инициализация членов класса. В данном случае было использовано два конструктора **public Book**: один – пустой, а второй наполняет поля класса начальными значениями, которые передаются через его параметры.

Поскольку имена параметров и имена полей **name**, **author**, **year** в данном случае совпадают, то мы используем ключевое слово **this**. Это ключевое слово представляет ссылку на текущий экземпляр класса. Поэтому в выражении **this.name = name;** первая часть **this.name** означает, что **name** – это поле текущего класса, а не название параметра **name**. Если бы параметры и поля назывались по-разному, то использование слова **this** было бы необязательно.

Теперь используем класс в программе. Создадим новый проект, а затем нажмем правой кнопкой мыши на название проекта в окне **Solution Explorer** и в появившемся меню выберем пункт **Class**.

В появившемся диалоговом окне дадим новому классу имя **Book** и нажмем кнопку **Add** (Добавить). В проект будет добавлен новый файл **Book.cs**, содержащий класс **Book**.

Изменим в этом файле код класса **Book**:

```
class Book
{
    public string name;
    public string author;
    public int year;
    public Book()
    {
        name = "неизвестно";
        author = "неизвестно";
        year = 0;
    }
    public Book(string name, string author, int year)
    {
        this.name = name;
        this.author = author;
        this.year = year;
    }
    public void GetInformation()
    {
        Console.WriteLine("Книга '{0}' (автор {1}) была из-
дана в {2} году", name, author, year);
    }
}
```

Теперь перейдем к коду файла **Program.cs** и изменим метод **Main** класса **Program** следующим образом:

```
class Program
{
    static void Main(string[] args)
    {
        Book b1 = new Book("Война и мир", "Л. Н. Толстой",
1869);
        b1.GetInformation();

        Book b2 = new Book();
        b2.GetInformation();

        Console.ReadLine();
    }
}
```

Если запустить код на выполнение, то на консоль выведется информация о книгах **b1** и **b2**. Следует обратить внимание, что для того чтобы создать новый объект с использованием конструктора, нам надо использовать ключевое слово **new**. Оператор **new** создает объект класса и выделяет для него область в памяти.

Для класса **Book** установим последовательно значения для всех трех его полей:

```
Book b1 = new Book();
b1.name = "Война и мир";
b1.author = "Л. Н. Толстой";
b1.year = 1869;

b1.GetInformation();
```

Но можно также использовать инициализатор объектов:

```
Book b2 = new Book();
b2 = new Book { name = "Отцы и дети", author = "И. С.
Тургенев", year = 1862 };
b2.GetInformation();
```

С помощью инициализатора объектов можно присваивать значения всем доступным полям и свойствам объекта в момент создания без явного вызова конструктора.

Все члены класса – поля, методы, свойства – имеют модификаторы доступа. Модификаторы доступа позволяют задать допустимую область видимости для членов класса, т. е. контекст, в котором можно употреблять данную переменную или метод. При объявлении полей класса **Book** использовался модификатор **public**, который означает, что все объявленные за ним члены класса общедоступны из любого места в коде, а также из других программ и сборок. Также в C# применяются и другие модификаторы доступа:

- **private** означает закрытый класс или член класса и представляет полную противоположность модификатору **public**, доступны только из кода в том же классе или контексте;
- **protected** означает, что член класса доступен из любого места в текущем классе или в производных классах;
- **internal** означает, что класс и члены класса доступны из любого места кода в той же сборке, но недоступны для других программ и сборок, как в случае с модификатором **public**.

- **protected internal** – совмещает функционал двух модификаторов. Классы и члены класса с таким модификатором доступны из текущей сборки и из производных классов.

Объявление полей класса без модификатора доступа равнозначно их объявлению с модификатором **private**. Классы, объявленные без модификатора, по умолчанию имеют доступ **internal**.

Рассмотрим это на примере создания класса **State**:

```
public class State
{
    int a; // все равно, что private int a;
// поле доступно только из текущего класса
    private int b;
// доступно из текущего класса и производных классов
    protected int c;
    internal int d; // доступно в любом месте программы
// доступно в любом месте программы
// и из классов-наследников
    protected internal int e;
// доступно в любом месте программы,
// а также для других программ и сборок
    public int f;

    private void Display_f()
    {
        Console.WriteLine("Переменная f = {0}", f);
    }

    public void Display_a()
    {
        Console.WriteLine("Переменная a = {0}", a);
    }

    internal void Display_b()
    {
        Console.WriteLine("Переменная b = {0}", b);
    }

    protected void Display_e()
    {
        Console.WriteLine("Переменная e = {0}", e);
    }
}
```

Так как класс **State** объявлен с модификатором **public**, он будет доступен из любого места программы, а также из других программ и сборок. Класс **State** имеет пять полей для каждого уровня доступа и одну переменную без модификатора, которая является закрытой по умолчанию.

Также имеются четыре метода, которые будут выводить значения полей класса на экран. Обратите внимание, что поскольку все модификаторы позволяют использовать члены класса внутри данного класса, то и все переменные класса, включая закрытые, доступны всем его методам, так как все находятся в контексте класса **State**.

Рассмотрим на примере возможность использования переменных класса **State** в методе **Main** класса **Program**:

```
class Program
{
  static void Main(string[] args)
  {
    State state1 = new State();

    // присвоить значение переменной a не получится,
   // так как она закрытая и класс Program ее не видит
   // И данную строку среда подчеркнет как неправильную

        state1.a = 4; //Ошибка, получить доступ нельзя

        // то же самое относится и к переменной b
        state1.b = 3; // Ошибка, получить доступ нельзя

   // присвоить значение переменной c не получится,
   // так как класс Program
   // не является классом-наследником класса State
        state1.c = 1; // Ошибка, получить доступ нельзя

        // переменная d с модификатором internal
        //  доступна из любого места программы
        // поэтому спокойно присваиваем ей значение
        state1.d = 5;

          // переменная e так же доступна
          // из любого места программы
        state1.e = 8;

        // переменная f общедоступна
        state1.f = 8;

        // Попробуем вывести значения переменных
```

```
    // Так как этот метод объявлен как private,
    // можно использовать его только внутри класса State
         state1.Display_f() ;

          // Ошибка, получить доступ нельзя
         // Так как этот метод объявлен как protected,
         // а класс Program не является
         // наследником класса State
         state1.Display_e();
            // Ошибка, получить доступ нельзя

         // Общедоступный метод
         state1.Display_a();

         // Метод доступен из любого места программы
         state1.Display_b();

         Console.ReadLine();

     }
}
```

Таким образом, установили только переменные **d**, **e** и **f**, так как их модификаторы позволяют использование в данном контексте. И оказались доступны только два метода: **state1.Display_a()** и **state1.Display_b()**. Однако, так как значения переменных **a** и **b** не были установлены, то эти методы выведут нули, так как значение переменных типа **int** по умолчанию инициализируются нулями.

Несмотря на то, что модификаторы **public** и **internal** похожи по своему действию, они имеют большое отличие. Классы и члены класса с модификатором **public** также будут доступны и другим программам, если данные класса поместить в динамическую библиотеку **dll** и потом использовать ее в этих программах. Благодаря такой системе модификаторов доступа можно скрывать некоторые моменты реализации класса от других частей программы. Такое сокрытие называется инкапсуляцией.

Кроме обычных методов в языке C# предусмотрены специальные методы доступа – свойства. Они обеспечивают простой доступ к полям класса и помогают узнать их значение или выполнить его установку.

Стандартное описание свойства имеет следующий синтаксис:

```
[модификатор_доступа] возвращаемый_тип произволь-
ное_название
{
    // код свойства
}
```

Например:

```
class Person
{
    private string name;

    public string Name
    {
        get
        {
            return name;
        }

        set
        {
            name = value;
        }
    }
}
```

В приведенном примере есть закрытое поле **name** и общедоступное свойство **Name.** Они имеют практически одинаковое название за исключением регистра, но это не более чем стиль, названия могут быть произвольные и не обязательно должны совпадать.

Через это свойство мы можем управлять доступом к переменной **name**. Стандартное определение свойства содержит блоки **get** и **set.** Блок **get** – возвращает значение поля, а блок **set** – устанавливает. Параметр **value** представляет передаваемое значение.

Возможно также использовать данное свойство следующим образом:

```
Person p = new Person();

// Устанавливаем свойство - срабатывает блок Set
// значение "Tom" и есть передаваемое в свойство value
p.Name = "Tom";

// Получаем значение свойства и присваиваем его переменной
- срабатывает блок Get
string personName = p.Name;
```

Возможно, может возникнуть вопрос, зачем нужны свойства, если мы можем в данной ситуации обходиться обычными полями класса? Свойства позволяют вложить дополнительную логику, которая может быть необходима,

например, при присвоении переменной класса какого-либо значения. Пусть надо установить проверку по возрасту:

```
class Person
{
    private int age;

    public int Age
    {
        set
        {
            if (value < 18)
            {
                Console.WriteLine("Возраст должен быть
больше 18");
            }
            else
            {
                age = value;
            }
        }
        get { return age; }
    }
}
```

Блоки **set** и **get** не обязательно одновременно должны присутствовать в свойстве. Например, можно закрыть свойство от установки, чтобы только получать значение. Для этого опускаем блок **set.** И наоборот, можно удалить блок **get**, тогда можно будет только установить значение, но нельзя получить:

```
class Person
{
    private string name;
    // свойство только для чтения
    public string Name
    {
        get
        {
            return name;
        }
    }

    private int age;
    // свойство только для записи
    public int Age
    {
```

```
        set
        {
            age = value;
        }
    }
}
```

Модификаторы доступа можно применять не только ко всему свойству, но и к отдельным блокам – либо **get**, либо **set**. При этом можно применить модификатор только к одному из блоков:

```
class Person
{
    private string name;

    public string Name
    {
        get
        {
            return name;
        }

        private set
        {
            name = value;
        }
    }
    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

Теперь закрытый блок **set** можно будет использовать только в данном классе, но никак не в другом:

```
Person p = new Person("Tom", 24);
// Ошибка - set объявлен с модификатором private
//p.Name = "John";
Console.WriteLine(p.Name);
```

Через свойства устанавливается доступ к приватным переменным класса. Подобное сокрытие состояния класса от вмешательства извне представляет механизм инкапсуляции, который является одной из ключевых концепций

объектно-ориентированного программирования. Применение модификаторов доступа типа **private** защищает переменную от внешнего доступа. Для управления доступом во многих языках программирования используются специальные методы, геттеры и сеттеры. В C# их роль, как правило, выполняют свойства.

Например, есть некоторый класс **Account**, в котором определено поле **sum**, представляющее сумму:

```
class Account
{
    public int sum;
}
```

Поскольку переменная **sum** является публичной, то в любом месте программы мы можем получить к ней доступ и изменить ее, в том числе установить какое-либо недопустимое значение, например отрицательное. Вряд ли подобное поведение является желательным. Поэтому применяется инкапсуляция для ограничения доступа к переменной **sum** и сокрытию ее внутри класса:

```
class Account
{
    private int sum;
    public int Sum
    {
        get {return sum;}
        set
        {
            if (value > 0)
            {
                sum=value;
            }
        }
    }
}
```

Свойства управляют доступом к полям класса. Однако, если полей десять и более, то определять каждое и писать для него однотипное свойство было бы утомительно. Поэтому, начиная с версии **.NET 4.0,** в фреймворк были добавлены автоматические свойства, которые имеют сокращенное объявление:

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public Person(string name, int age)
    {
        Name = name;
        Age = age;
    }
}
```

На самом деле тут также создаются поля для свойств, только их создает не программист в коде, а компилятор автоматически генерирует их при компиляции.

С одной стороны, автоматические свойства довольно удобны. С другой стороны, стандартные свойства имеют ряд преимуществ: например, они могут инкапсулировать дополнительную логику проверки значения; нельзя создать автоматическое свойство только для записи или чтения, как в случае со стандартными свойствами.

Ранее, для того чтобы использовать какой-нибудь класс и его методы, устанавливать и получать его поля, создавали его объект. Однако если данный класс имеет статические методы, то чтобы получить к ним доступ, необязательно создавать объект этого класса. Например, создадим новый класс **Algorithm** и добавим в него две функции для вычислений числа Фибоначчи и факториала:

```
class Algorithm
{
    public static double pi = 3.14;

    public static int Factorial(int x)
    {
        if (x == 1)
        {
            return 1;
        }
        else
        {
            return x * Factorial(x - 1);
```

```
        }
    }

    public static int Fibonachi(int x)
    {
        if (x == 0)
        {
            return 1;
        }
        if (x == 1)
        {
            return 1;
        }
        else
        {
            return Fibonachi(x - 1) + Fibonachi(x - 2);
        }
    }
}
```

Ключевое слово **static** при определении переменной и методов указывает, что данные члены будут доступны для всего класса, т. е. будут статическими, используем их в программе:

```
int num1 = Algorithm.Factorial(5);
int num2 = Algorithm.Fibonachi(5);

Algorithm.pi = 3.14159;
```

При использовании статических членов класса необязательно создавать экземпляр класса, можно обратиться к ним напрямую.

Для статических полей будет создаваться участок в памяти, который будет общим для всех объектов класса. Например, если создать два объекта класса **Algorithm**, то оба этих объекта будут хранить ссылку на один участок в памяти, где хранится значение **pi:**

```
Algorithm algorithm1 = new Algorithm();
Algorithm algorithm2 = new Algorithm();
```

Нередко статические поля применяются для хранения счетчиков. Например, пусть есть класс **State**, и нужен счетчик, который позволял бы узнать, сколько объектов **State** создано:

```
class State
{
    private static int counter = 0;
    public State()
    {
        counter++;
    }

    public static void DisplayCounter()
    {
        Console.WriteLine("Создано {0} объектов State",
counter);
    }
}
class Program
{
    static void Main(string[] args)
    {
        State state1 = new State();
        State state2 = new State();
        State state3 = new State();
        State state4 = new State();
        State state5 = new State();

        State.DisplayCounter(); // 5
        Console.Read();
    }
}
```

Кроме обычных конструкторов у класса также могут быть статические конструкторы. Статические конструкторы выполняются при самом первом создании объекта данного класса или первом обращении к его статическим членам, если таковые имеются:

```
class State
{
    static State()
    {
        Console.WriteLine("Создано первое государство");
    }
}
class Program
{
    static void Main(string[] args)
    {
        State s1 = new State(); // здесь сработает статиче-
ский конструктор
```

```
        State s2 = new State();

        Console.ReadLine();
    }
}
```

Наследование **inheritance** является одним из ключевых моментов ООП. Благодаря наследованию можно расширить функциональность уже существующих классов за счет добавления нового функционала или изменения старого. Пусть есть класс **Person**, описывающий отдельного человека:

```
class Person
{
    private string _firstName;
    private string _lastName;

    public string FirstName
    {
        get { return _firstName; }
        set { _firstName = value; }
    }
    public string LastName
    {
        get { return _lastName; }
        set { _lastName = value; }
    }

    public void Display()
    {
        Console.WriteLine(FirstName + " " + LastName);
    }
}
```

Но вдруг потребовался класс, описывающий сотрудника предприятия – класс **Employee**. Поскольку этот класс будет реализовывать тот же функционал, что и класс **Person**, так как сотрудник также человек, то было бы рационально сделать класс **Employee** производным или наследником от класса **Person**, который, в свою очередь, называется базовым классом или родителем:

```
class Employee : Person
{

}
```

После двоеточия мы указываем базовый класс для данного класса. Для класса **Employee** базовым является **Person**, и поэтому класс **Employee** наследует все те же свойства, методы, поля, которые есть в классе **Person**.

Таким образом, наследование реализует отношение **is-a**, что значит «является»:

```
static void Main(string[] args)
{
    Person p = new Person { FirstName = "Bill", LastName
= "Gates" };
    p.Display();
    p = new Employee { FirstName = "Denis", LastName =
"Ritchi" };
    p.Display();
    Console.Read();
}
```

И поскольку объект класса **Employee** является также и объектом класса **Person**, то можно так определить переменную:

```
Person p = new Employee().
```

Все классы по умолчанию могут наследоваться. Однако здесь есть ряд ограничений:

1. Не поддерживается множественное наследование, класс может наследоваться только от одного класса. Хотя проблема множественного наследования реализуется с помощью концепции интерфейсов.

2. При создании производного класса надо учитывать тип доступа к базовому классу – тип доступа к производному классу должен быть таким же, как и у базового класса, или более строгим. Таким образом, если базовый класс у нас имеет тип доступа **internal**, то производный класс может иметь тип доступа **internal** или **private**, но не **public**.

3. Если класс объявлен с модификатором **sealed**, то от этого класса нельзя наследовать и создавать производные классы. Например, следующий класс не допускает создание наследников:

```
sealed class Admin
{
}
```

Вернемся к классам **Person** и **Employee**. Хотя **Employee** наследует весь функционал от класса **Person**, посмотрим, что будет в следующем случае:

```
class Employee : Person
{
    public void Display()
    {
        Console.WriteLine(_firstName);
    }
}
```

Этот код не сработает и выдаст ошибку, так как переменная **_firstName** объявлена с модификатором **private** и поэтому доступ к ней имеет только класс **Person**. Но зато в классе **Person** определено общедоступное свойство **FirstName**, которое можно использовать, поэтому следующий код будет работать нормально:

```
class Employee : Person
{
    public void Display()
    {
        Console.WriteLine(FirstName);
    }
}
```

Таким образом, производный класс может иметь доступ только к тем членам базового класса, которые определены с модификаторами **public**, **internal**, **protected** и **protected internal**.

Теперь добавим в классы **Person** и **Employee** конструкторы:

```
class Person
{
    public string FirstName { get;  set; }
    public string LastName {get; set; }

    public Person(string fName, string lName)
    {
        FirstName = fName;
        LastName = lName;
    }

    public void Display()
    {
        Console.WriteLine(FirstName + " " + LastName);
    }
}

class Employee : Person
{
    public string Company { get; set; }
```

```
    public Employee(string fName, string lName, string comp)
            :base(fName, lName)
    {
        Company = comp;
    }
}
```

Класс **Person** имеет стандартный конструктор, который устанавливает два свойства. Поскольку класс **Employee** наследует и устанавливает те же свойства, что и класс **Person**, то логично было бы не писать каждый раз код установки, а вызвать соответствующий код класса **Person**. К тому же свойств, которые надо установить, и параметров может быть гораздо больше.

С помощью ключевого слова **base** можно обратиться к базовому классу. В нашем случае в конструкторе класса **Employee** надо установить имя, фамилию и компанию. Имя и фамилия передаются на установку в конструктор базового класса **Person** с помощью выражения **base(fName, lName)**:

```
static void Main(string[] args)
{
    Person p = new Person("Bill", "Gates");
    p.Display();
    Employee emp = new Employee ("Tom", "Simpson", "Mi-
crosoft");
    emp.Display();
    Console.Read();
}
```

Задание на лабораторную работу

Постройте иерархию классов в соответствии с вариантом задания:

Вариант	Задание
1	Студент, преподаватель, персона, заведующий кафедрой
2	Служащий, персона, рабочий, инженер
3	Рабочий, кадры, инженер, администрация
4	Деталь, механизм, изделие, узел
5	Организация, страховая компания, нефтегазовая компания, завод
6	Журнал, книга, печатное издание, учебник
7	Тест, экзамен, выпускной экзамен, испытание
8	Место, область, город, мегаполис
9	Игрушка, продукт, товар, молочный продукт
10	Квитанция, накладная, документ, счет
11	Автомобиль, поезд, транспортное средство, экспресс
12	Двигатель, дизель, двигатели внутреннего сгорания и реактивный

Вариант	Задание
13	Республика, монархия, королевство, государство
14	Млекопитающее, парнокопытное, птица, животное
15	Товар, велосипед, горный велосипед, самокат
16	Лев, дельфин, птица, синица, животное
17	Музыкант, персона, студент, гитарист
18	Печатное издание, газета, книга, периодика
19	Корабль, пароход, парусник, корвет
20	Стихотворение, стиль изложения, рифма, проза
21	Поселок, область, район, город
22	Грузовик, автомобиль, легковое авто, транспорт
23	Спорт, футбол, хобби, музыка
24	Молоток, инструмент, гитара, звук
25	Окружность, геометрическая фигура, линия, заливка

Порядок выполнения работы

1. Спроектируйте абстракции и представьте иерархию классов в виде схемы.

2. Разработайте конструкторы, атрибуты и методы для каждого из определяемых классов.

3. Реализуйте программу на языке C# в соответствии с вариантом исполнения, используя экземпляры описанных классов.

4. Примените и объясните необходимость использования принципа инкапсуляции.

5. Подготовьте документальный отчет, содержащий:

1) таблицу «Наименование сервиса (задачи, подзадачи, модуля)»;

2) схему алгоритма решения задачи;

3) код программы на исходном языке программирования с комментариями;

4) таблицу устранения ошибок;

5) предложения по модификации алгоритмов, кода.

Контрольные вопросы

1. Что понимается под термином «класс»?

2. Какие элементы определяются в составе класса?

3. Каково соотношение понятий «класс» и «объект»?

4. Что понимается под термином «члены класса»?

5. Какие члены класса Вам известны?

6. Какие члены класса содержат код?

7. Какие члены класса содержат данные?

8. Перечислите пять разновидностей членов класса специфичных для языка C#.

9. Что понимается под термином «конструктор»?

10. Сколько конструкторов может содержать класс языка C#?

11. Приведите синтаксис описания класса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

12. Какие модификаторы типа доступа Вам известны?

13. В чем заключаются особенности доступа членов класса с модификатором **public**?

14. В чем заключаются особенности доступа членов класса с модификатором **private**?

15. В чем заключаются особенности доступа членов класса с модификатором **protected**?

16. В чем заключаются особенности доступа членов класса с модификатором **internal**?

17. Какое ключевое слово языка C# используется при создании объекта?

18. Приведите синтаксис создания объекта в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

19. В чем состоит назначение конструктора?

20. Каждый ли класс языка C# имеет конструктор?

21. Какие умолчания для конструкторов приняты в языке C#?

22. Каким значением инициализируются по умолчанию переменные ссылочного типа?

23. В каком случае по умолчанию не используется конструктор класса?

24. Приведите синтаксис конструктора класса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

25. Что понимается под термином «деструктор»?

26. В чем состоит назначение деструктора?

27. Приведите синтаксис деструктора класса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

28. Что понимается под термином «наследование»?

29. Что общего имеет дочерний класс с родительским?

30. В чем состоит различие между дочерним и родительским классами?

Лабораторная работа №2

ПОЛИМОРФИЗМ

Цель работы: изучить механизмы реализации полиморфизма в C# и ознакомиться с основными подходами при использовании интерфейсов.

Краткие теоретические сведения

Полиморфизм является третьим ключевым аспектом объектно-ориентированного программирования и предполагает способность к изменению функционала, унаследованного от базового класса, и определение полиморфного интерфейса в базовом классе – набор членов класса, которые могут быть переопределены в классе-наследнике. Методы, которые надо переопределить, в базовом классе помечаются модификатором **virtual** и называются виртуальными. Они и представляют полиморфный интерфейс. Также частью полиморфного интерфейса могут быть абстрактные члены класса.

При определении класса-наследника и наследовании методов базового класса мы можем выбрать одну из следующих стратегий: обычное наследование всех членов базового класса, переопределение членов базового класса или скрытие членов базового класса в классе-наследнике.

Стратегия обычного наследования довольно проста. Допустим, есть следующая пара классов **Person** и **Employee**:

```
class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
    public Person(string lName, string fName)
    {
        FirstName = fName;
        LastName = lName;
    }
    public virtual void Display()
    {
        Console.WriteLine(FirstName + " " + LastName);
    }
}

class Employee : Person
{
    public string Company { get; set; }
    public Employee(string lName, string fName, string comp)
            :base(fName, lName)
    {
        Company = comp;
    }
}
```

В базовом классе **Person** метод **Display()** определен с модификатором **virtual**, поэтому данный метод может быть переопределен. Но класс **Employee** наследует его как есть:

```
class Program
{
    static void Main(string[] args)
    {
        Person p1 = new Person("Bill", "Gates");
        p1.Display(); // вызов метода Display из класса
Person

        Person p2 = new Employee("Tom", "Johns",
"UnitBank");
        p2.Display(); // вызов метода Display из класса
Person

        Employee p3 = new Employee("Sam", "Toms",
"CreditBank");
        p3.Display(); // вызов метода Display из класса
Person
        Console.Read();
    }
}
```

Консольный вывод:

```
Bill Gates
Tom Johns
Sam Toms
```

Переопределение членов базового класса в классе-наследнике предполагает использование ключевого слова **override**:

```
class Employee : Person
{
    public string Company { get; set; }
    public Employee(string lName, string fName, string comp)
            :base(fName, lName)
    {
        Company = comp;
    }

    public override void Display()
    {
        Console.WriteLine(FirstName + " " + LastName + " ра-
ботает в компании "+ Company);
    }
}
```

Класс **Person** остается тем же, метод **Display** также объявляется виртуальным. В этом случае поведение объекта **Employee** изменится:

```
Person p1 = new Person("Bill", "Gates");
p1.Display(); // вызов метода Display из класса Person

Person p2 = new Employee("Tom", "Johns", "UnitBank");
p2.Display(); // вызов метода Display из класса Employee

Employee p3 = new Employee("Sam", "Toms", "CreditBank");
p3.Display(); // вызов метода Display из класса Employee
```

Консольный вывод:

```
Bill Gates
Tom Johns работает в компании UnitBank
Sam Toms работает в компании CreditBank
```

При скрытии членов базового класса в классе-наследнике можно просто определить в нем метод с тем же именем, без переопределения, используя слово **override**:

```
class Employee : Person
{
    public string Company { get; set; }
    public Employee(string lName, string fName, string comp)
            :base(fName, lName)
    {
        Company = comp;
    }

    public new void Display()
    {
        Console.WriteLine(FirstName + " " + LastName + " работает в компании "+ Company);
    }
}
```

В этом случае метод **Display()** в **Employee** скрывает этот же метод из класса **Person**. Чтобы явно скрыть метод из базового класса, используется ключевое слово **new**, хотя в принципе оно необязательно, по умолчанию система это делает неявно.

Приведем пример использования ключевого слова **new** в программе:

```
Person p1 = new Person("Bill", "Gates");
p1.Display(); // вызов метода Display из класса Person

Person p2 = new Employee("Tom", "Johns", "UnitBank");
p2.Display(); // вызов метода Display из класса Person
```

```
Employee p3 = new Employee("Sam", "Toms", "CreditBank");
p3.Display(); // вызов метода Display из класса Employee
```

Консольный вывод:

```
Bill Gates
Tom Johns
Sam Toms работает в компании CreditBank
```

Следует обратить внимание на различия в вызовах метода **Display** из класса **Person**.

Кроме конструкторов, с помощью ключевого слова **base** можно обратиться к другим членам базового класса. Таким образом, вызов **base.Display()** будет обращением к методу **Display()** в классе **Person**:

```
class Employee : Person
{
    public string Company { get; set; }

    public Employee(string lName, string fName, string comp)
            :base(fName, lName)
    {
        Company = comp;
    }

    public override void Display()
    {
        base.Display();
        Console.WriteLine("Место работы : " + Company);
    }
}
```

Иногда возникает необходимость создать один и тот же метод, но с разным набором параметров. И в зависимости от имеющихся параметров применять определенную версию метода. Допустим, что есть следующий класс **State**:

```
class State
{
    public string Name { get; set; } // название
    public int Population { get; set; } // население
    public double Area { get; set; } // площадь
}
```

Необходимо определить метод для нападения на другое государство – метод **Attack**. Первая реализация этого метода будет принимать в качестве параметра объект **State** – это государство, на которое нападают:

```
public void Attack(State enemy)
{ }
```

Предположим, что надо определить версию данного метода, где будет указано не только государство, но и количество войск. Тогда можно будет просто добавить вторую версию данного метода:

```
public void Attack(State enemy)
{
    // здесь код метода
}
public void Attack(State enemy, int army)
{
    // здесь код метода
}
```

Наряду с методами возможно также перегружать операторы. При этом необходимо указывать модификаторы **public static**, так как перегружаемый оператор будет использоваться для всех объектов данного класса. После модификаторов идет название возвращаемого типа, и только после него ключевое слово **operator** и лишь затем название оператора и параметры:

```
public static возвращаемый_тип operator оператор(параметры)
```

Рассмотрим на примере. Есть класс **State** – государство. Необходимо объединить несколько государств. В этом случае задачу сможет облегчить перегрузка **operator +**. Кроме того, используем операторы сравнения «>» и «<», с помощью которых будем сравнивать два государства:

```
class State
{
    public string Name { get; set; } // название
    public int Population { get; set; } // население
    public double Area { get; set; } // площадь

    public static State operator +(State s1, State s2)
    {
        string name = s1.Name;
        int people = s1.Population+s2.Population;
        double area = s1.Area+s2.Area;

        // возвращаем новое объединенное государство
        return new State { Name = name, Area = area, Popula-
tion = people };
    }

    public static bool operator <(State s1, State s2)
    {
```

```
        if (s1.Area < s2.Area)
        {
            return true;
        }
        else
        {
            return false;
        }
    }

    public static bool operator >(State s1, State s2)
    {
        if (s1.Area > s2.Area)
        {
            return true;
        }
        else
        {
            return false;
        }
    }
}
```

Поскольку все перегруженные операторы бинарные, т. е. проводятся над двумя объектами, то для каждой перегрузки предусмотрено по два параметра. Теперь используем перегруженные операторы в программе:

```
static void Main(string[] args)
{
    State s1 = new State{ Name = "State1", Area = 300, Popu-
lation = 100 };

    State s2 = new State{ Name = "State2", Area = 200, Popu-
lation = 70 };

    if (s1 > s2)
    {
        Console.WriteLine("Государство s1 больше государст-
ва s2");
    }
    else if (s1 < s2)
    {
        Console.WriteLine("Государство s1 меньше государст-
ва s2");
    }
    else
    {
        Console.WriteLine("Государства s1 и s2 равны");
```

```
    }

    State s3 = s1 + s2;
    Console.WriteLine("Название государства : {0}",
s3.Name);
    Console.WriteLine("Площадь государства : {0}",
s3.Area);
    Console.WriteLine("Население государства : {0}",
s3.Population);

    Console.ReadLine();
}
```

Следует учитывать, что не все операторы можно перегружать.

Кроме обычных классов в C# есть абстрактные классы. Абстрактный класс похож на обычный. Он также может иметь переменные, методы, конструкторы и свойства. Но нельзя создать объект или экземпляр абстрактного класса. Абстрактные классы лишь предоставляют базовый функционал для классов-наследников, а реализуют его производные классы.

При определении абстрактных классов используется ключевое слово **abstract**:

```
abstract class Human
{
    public int Length { get; set; }
    public double Weight { get; set; }
}
```

Кроме обычных, абстрактный класс может иметь абстрактные методы. Подобные методы также определяются с помощью ключевого слова **abstract** и не имеют никакого функционала:

```
public abstract void Display();
```

При этом производный класс обязан переопределить и реализовать все абстрактные методы и свойства, которые имеются в базовом абстрактном классе. При переопределении в производном классе такой метод также объявляется с модификатором **override**. Также следует учесть, что если класс имеет хотя бы одно абстрактное свойство или метод, то он должен быть определен как абстрактный.

Абстрактные методы так же, как и виртуальные, являются частью полиморфного интерфейса. Но если в случае с виртуальными методами говорим, что класс-наследник наследует реализацию, то в случае с абстрактными методами наследуется интерфейс, представленный этими абстрактными методами.

Зачем нужны абстрактные классы? Допустим, программе для банковского сектора необходимо определить три класса: **person,** который описывает человека; **employee** – сотрудника банка; **client** – клиента банка.

Очевидно, что классы **Employee** и **Client** будут производными от класса **Person**. И так как все объекты будут представлять либо сотрудника банка, либо клиента, то напрямую от класса **Person** создавать объекты не будем. Поэтому имеет смысл сделать его абстрактным.

```
abstract class Person
{
    public string FirstName { get;  set; }
    public string LastName {get; set; }

    public Person(string lName, string fName)
    {
        FirstName = fName;
        LastName = lName;
    }

    public abstract void Display();
}

class Client : Person
{
    public string Bank { get; set; }

    public Client(string lName, string fName, string comp)
        : base(fName, lName)
    {
        Bank = comp;
    }

    public override void Display()
    {
        Console.WriteLine(FirstName + " " + LastName + "
имеет счет в банке " + Bank);
    }
}
```

Анонимные типы позволяют создать объект с некоторым набором свойств без определения класса. Определяются они с помощью ключевого слова **var** и инициализатора объектов:

```
var user = new { Name = "Tom", Age = 34 };
Console.WriteLine(user.Name);
```

В данном случае **user** – это объект анонимного типа с двумя определенными свойствами **Name** и **Age**, которые можно использовать так же, как и у обычных объектов классов. Однако тут есть ограничение – свойства анонимных типов доступны только для чтения.

При этом во время компиляции компилятор сам будет создавать для него имя типа и использовать это имя при обращении к объекту. Нередко анонимные типы имеют имя наподобие **"<>f__AnonymousType0'2"**.

Для исполняющей среды CLR анонимные типы будут также, как и классы, представлять ссылочный тип.

Если в программе используется несколько объектов анонимных типов с одинаковым набором свойств, то для них компилятор создаст одно определение анонимного типа:

```
var user = new { Name = "Tom", Age = 34 };
var student = new { Name = "Alice", Age = 21 };
var manager = new { Name = "Bob", Age = 26, Company =
"Microsoft" };

Console.WriteLine(user.GetType().Name); //
<>f__AnonymousType0'2
Console.WriteLine(student.GetType().Name); //
<>f__AnonymousType0'2
Console.WriteLine(manager.GetType().Name); //
<>f__AnonymousType1'3
```

Здесь **user** и **student** будут иметь одно и то же определение анонимного типа. Однако подобные объекты нельзя преобразовать к каком-нибудь другому типу, например классу, даже если он имеет подобный набор свойств.

Зачем нужны анонимные типы? Иногда возникает задача использовать один тип в одном узком контексте или даже один раз. Создание класса для подобного типа может быть избыточным. Если необходимо добавить свойство, то это можно будет сделать сразу на месте анонимного объекта. В случае с классом придется изменять еще и класс, который может больше нигде

не использоваться. Типичная ситуация – получение результата выборки из базы данных: объекты используются только для получения выборки и больше нигде не используются, поэтому классы для них создавать было бы излишне. А вот анонимный объект прекрасно подходит для временного хранения выборки.

Задание на лабораторную работу

Постройте иерархию классов, используя абстрактный класс и интерфейс в соответствии с вариантом задания. Реализуйте пример использования полиморфизма методов.

Вариант	Задание
1	Студент, преподаватель, персона, заведующий кафедрой
2	Служащий, персона, рабочий, инженер
3	Рабочий, кадры, инженер, администрация
4	Деталь, механизм, изделие, узел
5	Организация, страховая компания, нефтегазовая компания, завод
6	Журнал, книга, печатное издание, учебник
7	Тест, экзамен, выпускной экзамен, испытание
8	Место, область, город, мегаполис
9	Игрушка, продукт, товар, молочный продукт
10	Квитанция, накладная, документ, счет
11	Автомобиль, поезд, транспортное средство, экспресс
12	Двигатель, дизель, двигатели внутреннего сгорания и реактивный
13	Республика, монархия, королевство, государство
14	Млекопитающее, парнокопытное, птица, животное
15	Товар, велосипед, горный велосипед, самокат
16	Лев, дельфин, птица, синица, животное
17	Музыкант, персона, студент, гитарист
18	Печатное издание, газета, книга, периодика
19	Корабль, пароход, парусник, корвет
20	Стихотворение, стиль изложения, рифма, проза
21	Поселок, область, район, город
22	Грузовик, автомобиль, легковое авто, транспорт
23	Спорт, футбол, хобби, музыка
24	Молоток, инструмент, гитара, звук
25	Окружность, геометрическая фигура, линия, заливка

Порядок выполнения работы

1. Измените, расширьте и опишите иерархию классов, используя:

1) описание и наследование классами как минимум трех интерфейсов;

2) виртуальный класс в качестве основы полиморфизма.

2. Покажите на примере одного из методов, присутствующих в каждом классе, свойство полиморфизма.

3. Подготовьте отчет, содержащий:

1) таблицу «Наименование сервиса (задачи, подзадачи, модуля)»;

2) схему алгоритма решения задачи;

3) код программы на исходном языке программирования с комментариями;

4) таблицу устранения ошибок;

5) предложения по модификации алгоритмов, кода.

Контрольные вопросы

1. Что понимается под термином «полиморфизм»?

2. В чем состоит основной принцип полиморфизма?

3. В чем состоит значение основного принципа полиморфизма?

4. Какие механизмы используются в языке C# для реализации концепции полиморфизма?

5. Что понимается под термином «виртуальный метод»?

6. Какое ключевое слово языка C# используется для определения виртуального метода?

7. В чем состоит особенность виртуальных методов в производных (дочерних) классах?

8. В какой момент трансляции программы осуществляется выбор версии виртуального метода?

9. Какие условия определяют выбор версии виртуального метода?

10. Какое ключевое слово (модификатор) языка C# используется для определения виртуального метода в базовом (родительском) классе?

11. Какое ключевое слово (модификатор) языка C# используется для определения виртуального метода в производном (дочернем) классе?

12. Какие модификаторы недопустимы для определения виртуальных методов?

13. Что означает термин «переопределенный метод»?

14. В какой момент трансляции программы осуществляется выбор вызываемого переопределенного метода?

15. Приведите синтаксис виртуального метода в общем виде.

Лабораторная работа №3

РАБОТА С ФАЙЛАМИ

Цель работы: изучить основные принципы и приемы разработки приложений, использующих файлы для хранения данных.

Краткие сведения по теме

Большинство задач в программировании так или иначе связаны с работой с файлами и каталогами. Иногда требуется прочитать текст из файла или наоборот произвести запись, удалить файл или целый каталог, не говоря уже о более комплексных задачах, таких как создание текстового редактора и других подобных.

Фреймворк **.NET** предоставляет большие возможности по управлению и манипуляции файлами и каталогами, которые по большей части сосредоточены в пространстве имен **System.IO**. Классы, расположенные в этом пространстве имен (такие как **Stream**, **StreamWriter**, **FileStream** и др.), позволяют управлять файловым вводом-выводом.

Работу с файловой системой начнем с самого верхнего уровня – с дисков. Для представления диска в пространстве имен **System.IO** имеется класс **DriveInfo**.

Этот класс имеет статический метод **GetDrives**, который возвращает имена всех логических дисков компьютера. Также он предоставляет ряд полезных свойств:

- **AvailableFreeSpace** указывает на объем доступного свободного места на диске в байтах;
- **DriveFormat** получает имя файловой системы;
- **DriveType** представляет тип диска;
- **IsReady** указывает на готовность диска (например, DVD-диск может быть не вставлен в дисковод);
- **Name** получает имя диска;
- **TotalFreeSpace** получает общий объем свободного места на диске в байтах;
- **TotalSize** получает общий размер диска в байтах;
- **VolumeLabel** получает или устанавливает метку тома.

Получим имена и свойства всех дисков на компьютере:

```
using System;
using System.Collections.Generic;
using System.IO;
```

```
namespace FileApp
{
    class Program
    {
        static void Main(string[] args)
        {
            DriveInfo[] drives = DriveInfo.GetDrives();

            foreach (DriveInfo drive in drives)
            {
                Console.WriteLine("Название: {0}",
drive.Name);
                Console.WriteLine("Тип: {0}",
drive.DriveType);
                if (drive.IsReady)
                {
                    Console.WriteLine("Объем диска: {0}",
drive.TotalSize);
                    Console.WriteLine("Свободное простран-
ство: {0}", drive.TotalFreeSpace);
                    Console.WriteLine("Метка: {0}",
drive.VolumeLabel);
                }
                Console.WriteLine();
            }

            Console.ReadLine();
        }
    }
}
```

Для работы с каталогами в пространстве имен **System.IO** предназначены сразу два класса: **Directory** и **DirectoryInfo**.

Класс **Directory** предоставляет ряд статических методов для управления каталогами:

- **CreateDirectory(path)** создает каталог по указанному пути **path**;
- **Delete(path)** удаляет каталог по указанному пути **path**;
- **Exists(path)** определяет, существует ли каталог по указанному пути **path** и если существует, возвращает **true**, если нет **false**;
- **GetDirectories(path)** получает список каталогов в каталоге **path**;
- **GetFiles(path)** получает список файлов в каталоге **path**;
- **Move(sourceDirName, destDirName)** перемещает каталог;
- **GetParent(path)** получает родительский каталог.

Класс **DirectoryInfo** предоставляет функциональность для создания, удаления, перемещения и других операций с каталогами. Во многом он похож на **Directory**. Некоторые из его свойств и методов:

- **Create()** создает каталог;
- **CreateSubdirectory(path)** создает подкаталог по указанному пути **path**;
- **Delete()** удаляет каталог;
- **Свойство Exists** определяет, существует ли каталог;
- **GetDirectories()** получает список каталогов;
- **GetFiles()** получает список файлов;
- **MoveTo(destDirName)** перемещает каталог;
- **Parent** получает родительский каталог;
- **Root** получает корневой каталог.

Рассмотрим на примерах применение этих классов.

Получение списка файлов и подкаталогов:

```
string dirName = "C:\\";

if (Directory.Exists(dirName))
{
    Console.WriteLine("Подкаталоги:");
    string[] dirs = Directory.GetDirectories(dirName);
    foreach (string s in dirs)
    {
        Console.WriteLine(s);
    }
    Console.WriteLine();
    Console.WriteLine("Файлы:");
    string[] files = Directory.GetFiles(dirName);
    foreach (string s in files)
    {
        Console.WriteLine(s);
    }
}
```

Обратите внимание на использование слешей в именах файлов. Если используется одинарный, то перед всем путем ставим знак **@**: @"C:\Program Files". Если используется двойной, то тогда: "C:\\".

Создание каталога:

```
string path = @"C:\SomeDir";
string subpath = @"program\avalon";
DirectoryInfo dirInfo = new DirectoryInfo(path);
if (!dirInfo.Exists)
{
    dirInfo.Create();
}
dirInfo.CreateSubdirectory(subpath);
```

Сначала необходимо проверить, существует директория с таким именем или нет: если нет, то создать ее будет нельзя и приложение выбросит ошибку. В итоге получаем следующий путь: "C:\SomeDir\program\avalon".

Получение информации о каталоге:

```
string dirName = "C:\\Program Files";

DirectoryInfo dirInfo = new DirectoryInfo(dirName);

Console.WriteLine("Название каталога: {0}", dirInfo.Name);
Console.WriteLine("Полное название каталога: {0}",
dirInfo.FullName);
Console.WriteLine("Время создания каталога: {0}",
dirInfo.CreationTime);
Console.WriteLine("Корневой каталог: {0}", dirInfo.Root);
```

Если просто применить метод **Delete** к непустой папке, в которой есть какие-нибудь файлы или подкаталоги, то приложение выбросит ошибку. Поэтому надо передать в метод **Delete** дополнительный параметр булевого типа, который укажет, что папку надо удалять со всем содержимым:

```
string dirName = @"C:\SomeFolder";

try
{
    DirectoryInfo dirInfo = new DirectoryInfo(dirName);
    dirInfo.Delete(true);
}
catch (Exception ex)
{
    Console.WriteLine(ex.Message);
}
```

или

```
string dirName = @"C:\SomeFolder";

Directory.Delete(dirName, true);
```

Перемещение каталога:

```
string oldPath = @"C:\SomeFolder";
string newPath = @"C:\SomeDir";
DirectoryInfo dirInfo = new DirectoryInfo(oldPath);
if (dirInfo.Exists && Directory.Exists(newPath) == false)
{
    dirInfo.MoveTo(newPath);
}
```

При перемещении каталога надо учитывать, что новый, в который необходимо перемесить все содержимое старого, не должен существовать.

Подобно паре **Directory/DirectoryInfo** для работы с файлами предназначена пара классов **File/FileInfo**. С их помощью можно создавать, удалять, перемещать файлы, получать их свойства и многое другое.

Некоторые полезные методы и свойства класса **FileInfo**:

- **CopyTo(path)** копирует файл в новое место по указанному пути **path**;
- **Create()** создает файл;
- **Delete()** удаляет файл;
- **MoveTo(destFileName)** перемещает файл в новое место;
- свойство **Directory** получает родительский каталог в виде объекта **DirectoryInfo**;
- свойство **DirectoryName** получает полный путь к родительскому каталогу;
- свойство **Exists** указывает, существует ли файл;
- свойство **Length** получает размер файла;
- свойство **Extension** получает расширение файла;
- свойство **Name** получает имя файла;
- свойство **FullName** получает полное имя файла.

Класс **File** реализует похожую функциональность с помощью статических методов:

- **Copy()** копирует файл в новое место;
- **Create()** создает файл;
- **Delete()** удаляет файл;
- **Move** перемещает файл в новое место;
- **Exists(file)** определяет, существует ли файл.

Получение информации о файле:

```
string path = @"C:\apache\hta.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
    Console.WriteLine("Имя файла: {0}", fileInf.Name);
    Console.WriteLine("Время создания: {0}",
fileInf.CreationTime);
    Console.WriteLine("Размер: {0}", fileInf.Length);
}
```

Удаление файла:

```
string path = @"C:\apache\hta.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
   fileInf.Delete();
   // альтернатива с помощью класса File
   // File.Delete(path);
}
```

Перемещение файла:

```
string path = @"C:\apache\hta.txt";
string newPath = @"C:\SomeDir\hta.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
   fileInf.MoveTo(newPath);
   // альтернатива с помощью класса File
   // File.Move(path, newPath);
}
```

Копирование файла:

```
string path = @"C:\apache\hta.txt";
string newPath = @"C:\SomeDir\hta.txt";
FileInfo fileInf = new FileInfo(path);
if (fileInf.Exists)
{
   fileInf.CopyTo(newPath, true);
   // альтернатива с помощью класса File
   // File.Copy(path, newPath, true);
}
```

Метод **CopyTo** класса **FileInfo** принимает два параметра: путь, по которому файл будет копироваться, и булевое значение, которое указывает, надо ли при копировании перезаписывать файл (если **true**, то файл при копировании перезаписывается). Если в качестве последнего параметра передать значение **false**, то если такой файл уже существует, приложение выдаст ошибку.

Метод **Copy** класса **File** принимает три параметра: путь к исходному файлу и тот, по которому файл будет копироваться, и булевое значение, указывающее, будет ли файл перезаписываться.

Класс **FileStream** представляет возможности по считыванию из файла и записи в него. Он позволяет работать как с текстовыми файлами, так и с бинарными.

Рассмотрим наиболее важные его свойства и методы:

• свойство **Length** возвращает длину потока в байтах;

• свойство **Position** возвращает текущую позицию в потоке;

• метод **long Seek(long offset, SeekOrigin origin)** устанавливает позицию в потоке со смещением на количество байт, указанных в параметре **offset**;

• метод **Write** записывает в файл данные из массива байтов, принимает три параметра **Write(byte[] array, int offset, int count)**: **array** – массив байтов, куда будут помещены считываемые из файла данные; **offset** – представляет смещение в байтах в массиве **array**, в который будут помещены считанные байты; **count** – максимальное число байтов, предназначенных для чтения, но если в файле находится меньшее количество байтов, то все они будут считаны;

• метод **Read** считывает данные из файла в массив байтов и возвращает количество успешно считанных байтов, принимает три параметра **int Read(byte[] array, int offset, int count)**: **array** – массив байтов, откуда данные будут записываться в файл; **offset** – смещение в байтах в массиве **array**, откуда начинается запись байтов в поток; **count** – максимальное число байтов, предназначенных для записи.

FileStream представляет доступ к файлам на уровне байтов, поэтому, например, если вам надо считать или записать одну или несколько строк в текстовый файл, то массив байтов надо преобразовать в строки, используя специальные методы. Поэтому для работы с текстовыми файлами применяются другие классы.

В то же время при работе с различными бинарными файлами, имеющими определенную структуру, **FileStream** может быть очень даже полезен для извлечения определенных порций информации и ее обработки.

В качестве примера рассмотрим считывание-запись в текстовый файл:

```
Console.WriteLine("Введите строку для записи в файл:");
string text = Console.ReadLine();

// запись в файл
using (FileStream fstream = new
FileStream(@"C:\SomeDir\noname\note.txt",
FileMode.OpenOrCreate))
{
    // преобразуем строку в байты
    byte[] array = Sys-
tem.Text.Encoding.Default.GetBytes(text);
    // запись массива байтов в файл
    fstream.Write(array, 0, array.Length);
    Console.WriteLine("Текст записан в файл");
}

// чтение из файла
using (FileStream fstream =
File.OpenRead(@"C:\SomeDir\noname\note.txt"))
{
    // преобразуем строку в байты
    byte[] array = new byte[fstream.Length];
    // считываем данные
    fstream.Read(array, 0, array.Length);
    // декодируем байты в строку
    string textFromFile = Sys-
tem.Text.Encoding.Default.GetString(array);
    Console.WriteLine("Текст из файла: {0}", textFromFile);
}

Console.ReadLine();
```

Разберем этот пример. И при чтении, и при записи используется оператор **using**. Не надо путать данный оператор с директивой **using**, которая подключает пространства имен в начале файла кода. Оператор **using** позволяет создавать объект в блоке кода, по завершению которого вызывается метод **Dispose** у этого объекта, и, таким образом, объект уничтожается. В данном случае в качестве такого объекта служит переменная **fstream**.

Объект **fstream** создается двумя разными способами: через конструктор и один из статических методов класса **File**.

Здесь в конструктор передается два параметра: путь к файлу и перечисление **FileMode**. Данное перечисление указывает на режим доступа к файлу и может принимать следующие значения:

- **Append**, если файл существует, то текст добавляется в конец файла. Если файла нет, то он создается. Файл открывается только для записи.
- **Create** создает новый файл и если такой файл уже существует, то он перезаписывается.
- **CreateNew** создает новый файл и если такой файл уже существует, то в приложение выбрасывает ошибку.
- **Open** открывает файл и если файл не существует, выбрасывается исключение.
- **Create** создает новый файл и если такой файл уже существует, то он перезаписывается.
- **OpenOrCreate** если файл существует, он открывается, если нет – создается новый.
- **Truncate** если файл существует, то он перезаписывается. Файл открывается только для записи.

Статический метод **OpenRead** класса **File** открывает файл для чтения и возвращает объект **FileStream**.

Конструктор класса **FileStream** также имеет ряд перегруженных версий, позволяющий более точно настроить создаваемый объект. Все эти версии можно посмотреть на Microsoft Developer Network (MSDN**)**.

При записи и при чтении применяется объект кодировки **Encoding.Default** из пространства имен **System.Text**. В данном случае использовали два его метода: **GetBytes** для получения массива байтов из строки и **GetString** – строки из массива байтов.

В итоге введенная строка записывается в файл **noname**. По сути это бинарный, не текстовый файл, хотя если в него записать только строку, то можно будет просмотреть его, открыв в текстовом редакторе. Однако, если записать в него случайные байты, то могут возникнуть проблемы с его пониманием:

```
fstream.WriteByte(13);
fstream.WriteByte(103);
```

Нередко бинарные файлы представляют определенную структуру, зная которую можно взять из файла нужную порцию информации или наоборот записать в конкретном месте файла определенный набор байтов. Например, в wav-файлах непосредственно звуковые данные начинаются с сорок четвертого байта, а до него идут различные метаданные – количество каналов аудио, частота дискретизации и т.д.

С помощью метода **Seek()** мы можем управлять положением курсора потока, начиная с которого производится считывание или запись в файл. Этот метод принимает два параметра: **offset** (смещение) и позиция в файле. Позиция в файле описывается тремя значениями:

- **SeekOrigin.Begin** – начало файла;
- **SeekOrigin.End** – конец файла;
- **SeekOrigin.Current** – текущая файла.

Курсор потока, с которого начинается чтение или запись, смещается вперед на значение **offset** относительно позиции, указанной в качестве второго параметра. Смещение может быть как отрицательным, тогда курсор сдвигается назад, так и положительным – курсор сдвигается вперед.

Рассмотрим на примере:

```
using System.IO;
using System.Text;

class Program
{
    static void Main(string[] args)
    {
        string text = "hello world";

        // запись в файл
        using (FileStream fstream = new
FileStream(@"D:\note.dat", FileMode.OpenOrCreate))
        {
            // преобразуем строку в байты
            byte[] input = Encoding.Default.GetBytes(text);
            // запись массива байтов в файл
            fstream.Write(input, 0, input.Length);
            Console.WriteLine("Текст записан в файл");

            // перемещаем указатель в конец файла, до конца
файла- пять байт
            fstream.Seek(-5, SeekOrigin.End); // минус 5
символов с конца потока
```

```
            // считываем четыре символов с текущей позиции
            byte[] output = new byte[4];
            fstream.Read(output, 0, output.Length);
            // декодируем байты в строку
            string textFromFile = Encod-
ing.Default.GetString(output);
            Console.WriteLine("Текст из файла: {0}",
textFromFile); // worl

            // заменим в файле слово world на слово house
            string replaceText = "house";
            fstream.Seek(-5, SeekOrigin.End); // минус 5
символов с конца потока
            input = Encoding.Default.GetBytes(replaceText);
            fstream.Write(input, 0, input.Length);

            // считываем весь файл
            // возвращаем указатель в начало файла
            fstream.Seek(0, SeekOrigin.Begin);
            output = new byte[fstream.Length];
            fstream.Read(output, 0, output.Length);
            // декодируем байты в строку
            textFromFile = Encod-
ing.Default.GetString(output);
            Console.WriteLine("Текст из файла: {0}",
textFromFile); // hello house
        }
        Console.Read();
    }
}
```

Консольный вывод:

```
Текст записан в файл
Текст из файла: world
Текст из файла: hello house
```

Вызов **fstream.Seek(-5, SeekOrigin.End)** перемещает курсор потока в конец файлов назад на пять символов, т. е. после записи в новый файл строки "**hello world**" курсор будет стоять на позиции символа "**w**".

После этого считываем четыре байта, начиная с символа "**w**". В данной кодировке один символ будет представлять один байт. Поэтому чтение четырех байтов будет эквивалентно чтению четырех символов: "**worl**".

Затем опять перемещаемся в конец файла, не доходя до конца пять символов (т. е. опять на позицию символа "**w**"), и осуществляем запись строки "**house**". Таким образом, строка "**house**" заменяет строку "**world**".

В примерах, рассмотренных выше, для закрытия потока применяется конструкция **using**. После того как все операторы и выражения в блоке **using** отработают, объект **FileStream** уничтожается. Однако, возможно выбрать и другой способ:

```
FileStream fstream = null;

try
{
    fstream = new FileStream(@"D:\note3.dat",
FileMode.OpenOrCreate);
    // операции с потоком
}
catch(Exception ex)
{

}
finally
{
    if (fstream != null)
        fstream.Close();
}
```

Если использовать конструкцию **using**, то надо явным образом вызвать метод **Close(): fstream.Close()**.

Класс **FileStream** не очень удобно применять для работы с текстовыми файлами. К тому же для этого в пространстве **System.IO** определены специальные классы: **StreamReader** и **StreamWriter**.

Класс **StreamReader** позволяет легко считывать весь текст или отдельные строки из текстового файла. Среди его методов можно выделить следующие:

• **Close** закрывает считываемый файл и освобождает все ресурсы;

• **Peek** возвращает следующий доступный символ, если символов больше нет, то возвращает 1;

• **ReadLine** считывает одну строку в файле;

• **ReadToEnd** считывает весь текст из файла;

• **Read** считывает и возвращает следующий символ в численном представлении и имеет перегруженную версию:

```
Read(char[] array, int index, int count)
```

где **array** – массив, куда считываются символы; **index** – индекс в массиве **array**, начиная с которого записываются считываемые символы; **count** – максимальное количество считываемых символов.

Считаем текст из файла различными способами:

```
string path= @"C:\SomeDir\hta.txt";

try
{
    Console.WriteLine("******считываем весь файл********");
    using (StreamReader sr = new StreamReader(path))
    {
        Console.WriteLine(sr.ReadToEnd());
    }

    Console.WriteLine();
    Console.WriteLine("******считываем построчно********");
    using (StreamReader sr = new StreamReader(path, Sys-
tem.Text.Encoding.Default))
    {
        string line;
        while ((line = sr.ReadLine()) != null)
        {
            Console.WriteLine(line);
        }
    }

    Console.WriteLine();
    Console.WriteLine("******считываем блоками********");
    using (StreamReader sr = new StreamReader(path, Sys-
tem.Text.Encoding.Default))
    {
        char[] array = new char[4];
        // считываем 4 символа
        sr.Read(array, 0, 4);

        Console.WriteLine(array);
    }
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
```

Как и в случае с классом **FileStream** здесь используется конструкция **using**.

В первом случае мы разом считываем весь текст с помощью метода **ReadToEnd()**.

Во втором случае считываем построчно через цикл **while,** присваивая сначала переменной **line** результат функции **sr.ReadLine()**, а затем проверяем, не равна ли она **null**:

```
while ((line = sr.ReadLine()) != null)
```

и когда объект **sr** дойдет до конца файла и больше строк не останется, то метод **sr.ReadLine()** будет возвращать **null**.

В третьем случае считываем в массив четыре символа.

Обратите внимание, что в последних двух случаях в конструкторе **StreamReader** указывалась кодировка **System.Text.Encoding.Default**. Свойство **Default** класса **Encoding** получает кодировку для текущей кодовой страницы ANSI. Также через другие свойства можно указать другие кодировки. Если кодировка не указана, то при чтении используется UTF8. Но иногда важно указывать кодировку, так как она может отличаться от UTF8, и тогда возможно получить некорректный вывод.

Для записи в текстовый файл используется класс **StreamWriter**. Свою функциональность он реализует через следующие методы:

• **Close –** закрывает записываемый файл и освобождает все ресурсы;

• **Flush** – записывает в файл оставшиеся в буфере данные и очищает буфер;

• **Write –** записывает в файл данные простейших типов, как **int**, **double**, **char**, **string** и т.д.;

• **WriteLine** – записывает данные, только после записи добавляет в файл символ окончания строки.

Рассмотрим пример записи в текстовый файл:

```
string readPath= @"C:\SomeDir\hta.txt";
string writePath = @"C:\SomeDir\ath.txt";

string text = "";
try
{
    using (StreamReader sr = new StreamReader(readPath, Sys-
tem.Text.Encoding.Default))
```

```
    {
        text=sr.ReadToEnd();
    }
    using (StreamWriter sw = new StreamWriter(writePath,
false, System.Text.Encoding.Default))
    {
        sw.WriteLine(text);
    }

    using (StreamWriter sw = new StreamWriter(writePath,
true, System.Text.Encoding.Default))
    {
        sw.WriteLine("Дозапись");
        sw.Write(4.5);
    }
}
catch (Exception e)
{
    Console.WriteLine(e.Message);
}
```

Сначала считываем файл в переменную **text**, а затем записываем эту переменную в файл, а затем через объект **StreamWriter** записываем в новый файл, используя один из конструкторов:

```
new StreamWriter(writePath, false, Sys-
tem.Text.Encoding.Default).
```

Здесь первый параметр передает путь к записываемому файлу, второй – представляет булевую переменную, определяющую, будет файл дозаписываться или перезаписываться. Если булева переменная равна **true,** то новые данные добавляются в конец к уже имеющимся, а если **false** – файл перезаписывается. Исходя из этого, получаем, что файл перезаписывается в первом случае, а во втором – дозаписывается. Третий параметр указывает кодировку, в которой записывается файл.

Задания на лабораторную работу

Реализуйте запросы, определив:

1) фамилии студентов, у которых две и более двоек за сессию, и удалить их (выведя сообщение);

2) институт, на котором на первом курсе наибольшее количество отличников;

3) курс, на котором исключено большее количество студентов;

4) институт с наибольшим количеством отличников;

5) полный список отличников с указанием института, группы и курса, где они учатся;

6) группу, где нет двоечников;

7) институт и курс, на котором средний бал не меньше 3,5;

8) фамилии студентов, у которых нет троек и двоек;

9) институт и группу, где наибольшее количество отличников;

10) фамилии студентов-отличников на третьем курсе;

11) предметы и перечень кафедр, на которых они присутствуют;

12) фамилии студентов, группу и институт, где средний балл составляет 4,5;

13) студентов первого курса, у которых три двойки и удалите их;

14) группы, в которых нет двоечников;

15) фамилии студентов-отличников на первом и втором курсах по всем институтам, средний балл по каждой группе и упорядочьте группы по нему;

16) институты, на которых нет двоечников;

17) фамилии студентов, которые не явились хотя бы на один экзамен (оценка 0) и удалите тех, у которых средний балл ниже 3;

18) институт, на котором на первом курсе наибольшее количество групп, где нет двоек;

19) курс с наибольшим количеством отличников;

20) институт, на котором на первом курсе наибольшее количество двоечников;

21) группы, в которых нет отличников;

22) полный список двоечников с указанием института, группы и курса, где они учатся;

23) фамилии студентов-отличников на втором курсе с указанием группы и института, где они учатся.

Порядок выполнения лабораторной работы

1. Создайте классы институтов (данные о курсах и группах), предметов и студентов.

2. Реализуйте возможность ввода, удаления, редактирования данных в массив (коллекцию).

3. Реализуйте вывод результата запроса в соответствии с вариантом исполнения в файл.

4. Представьте реализацию перечисленных действий в виде меню программы с возможностью их повторов до выбора пункта «Выход из программы».

5. Подготовьте отчет.

Лабораторная работа №4

РАСШИРЕННЫЕ ВОЗМОЖНОСТИ ПРОГРАММИРОВАНИЯ НА C#

Цель работы: познакомиться с расширенными возможностями языка программирования C#, такими как интерфейсы и делегаты.

Краткие теоретические сведения

Используя механизм наследования, воззможно дополнять и переопределять общий функционал базовых классов в классах-наследниках. Однако, напрямую можно наследовать только от одного класса, в отличие, например, от языка C++, где имеется множественное наследование.

В языке C# подобную проблему позволяют решить интерфейсы. Они играют важную роль в системе ООП. Интерфейсы позволяют определить некоторый функционал, не имеющий конкретной реализации. Затем этот функционал реализуют классы, применяющие данные интерфейсы.

Для определения интерфейса используется ключевое слово **interface.** Как правило, названия интерфейсов в C# начинаются с заглавной буквы «**I»,** например, **IComparable**, **IEnumerable**, однако это не обязательное требование, а больше стиль программирования. Интерфейсы также, как и классы, могут содержать свойства, методы и события, только без конкретной реализации.

Определим следующий интерфейс **IAccount**, который будет содержать методы и свойства для работы со счетом клиента. Для добавления интефрейса в проект нужно нажать правой кнопкой мыши на проект и в появившемся контекстном меню выбрать **Add → New Item** и в диалоговом окне добавления нового компонента выбрать **Interface.**

Изменим пустой код интерфейса **IAccount** на следующий:

```
interface IAccount
{
    // Текущая сумма на счету
    int CurrentSum { get; }
    // Положить деньги на счет
    void Put(int sum);
    // Взять со счета
    void Withdraw(int sum);
    // Процент начислений
    int Percentage { get; }
}
```

У интерфейса методы и свойства не имеют реализации, в этом они сближаются с абстрактными методами абстрактных классов. Сущность данного интерфейса проста: он определяет два свойства для текущей суммы денег на счете и ставки процента по вкладам и два метода для добавления денег на счет и их изъятия.

При объявлении интерфейса все его члены (методы и свойства) не имеют модификаторов доступа, но по умолчанию у них доступ **public**, так как целью является определение функционала для реализации его классом. Поэтому весь функционал должен быть открыт для реализации.

Применение интерфейса аналогично наследованию класса:

```
class Client : IAccount
{
// реализация методов и свойств интерфейса
}
```

Так как клиент обладает счетом, реализуем интерфейс в классе **Client**:

```
class Client : IAccount
{
    int _sum; // Переменная для хранения суммы
    int _percentage; // Переменная для хранения процента

    public string Name { get; set; }
    public Client(string name, int sum, int percentage)
    {
        Name = name;
        _sum = sum;
        _percentage = percentage;
    }

    public int CurrentSum
    {
        get { return _sum; }
    }

    public void Put(int sum)
    {
        _sum += sum;
    }

    public void Withdraw(int sum)
```

```
    {
        if (sum <= _sum)
        {
            _sum -= sum;
        }
    }

    public int Percentage
    {
        get { return _percentage; }
    }
    public void Display()
    {
        Console.WriteLine("Клиент " + Name + " имеет счет на
сумму " + _sum);
    }
}
```

Как и в случае с абстрактными методами абстрактного класса класс **Client** реализует все методы интерфейса. Поскольку все методы и свойства интерфейса являются публичными, то при их реализации в классе к ним можно применять только модификатор **public**. Поэтому если класс должен иметь метод с каким-то другим модификатором, например **protected**, то интерфейс не подходит для определения подобного метода.

Применение класса в программе:

```
Client client = new Client("Tom", 200, 10);
client.Put(30);
Console.WriteLine(client.CurrentSum); //230
client.Withdraw(100);
Console.WriteLine(client.CurrentSum); //130
```

Интерфейсы, как и классы, могут наследоваться:

```
interface IDepositAccount : IAccount
{
    void GetIncome(); // начисление процентов
}
```

При применении этого интерфейса класс **Client** должен будет реализовать как методы и свойства интерфейса **IDepositAccount**, так и базового интерфейса **IAccount.**

Задания на лабораторную работу

Реализуйте для иерархии из лабораторной работы №3 механизм интерфейсов, при этом один из классов должен реализовывать как минимум два интерфейса. Используйте для проверки всех методов данного класса многоадресный делегат. Подготовьте отчет о проделанной работе.

Контрольные вопросы

1. Что понимается под термином «интерфейс»?
2. Чем отличается синтаксис интерфейса от синтаксиса абстрактного класса?
3. Какое ключевое слово языка C# используется для описания интерфейса?
4. Поддерживают ли реализацию методы интерфейса?
5. Какие объекты языка C# могут быть членами интерфейсов?
6. Каким количеством классов может быть реализован интерфейс?
7. Может ли класс реализовывать множественные интерфейсы?
8. Необходима ли реализация методов интерфейса в классе, включающем этот интерфейс?
9. Какой модификатор доступа соответствует интерфейсу?
10. Допустимо ли явное указание модификатора доступа для интерфейса?
11. Приведите синтаксис интерфейса в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
12. Возможно ли создание ссылочной переменной интерфейсного типа?
13. Возможно ли наследование интерфейсов?
14. Насколько синтаксис наследования интерфейсов отличается от синтаксиса наследования классов?
15. Необходимо ли обеспечение реализации в иерархии наследуемых интерфейсов?
16. Что понимается под термином «делегат»?
17. В чем состоят преимущества использования делегатов?
18. В какой момент осуществляется выбор вызываемого метода в случае использования делегатов?
19. Что является значением делегата?
20. Какое ключевое слово языка C# используется для описания делегатов?
21. Приведите синтаксис делегата в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.
22. Возможно ли использование делегата для вызова метода, соответствующего подписи делегата?
23. Возможен ли вызов метода в том случае, если его подпись не соответствует подписи делегата?
24. Что понимается под термином «многоадресность»?
25. В чем состоит практическое значение многоадресности?

Лабораторная работа № 5

СОБЫТИЙНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Цель работы: ознакомить с механизмами событийно-ориентированного программирования на языке C#, такими как механизм обработки событий и исключительные ситуации.

Краткие теоретические сведения

Кроме свойств и методов классы и интерфейсы могут содержать делегаты и события. Делегаты представляют такие объекты, которые указывают на другие методы, т. е. делегаты – это указатели на методы. С помощью делегатов мы можем вызвать определенные методы в ответ на некоторые произошедшие действия. Таким образом, по своей сути делегаты раскрывают функционал функций обратного вызова.

Методы, на которые ссылаются делегаты, должны иметь те же параметры и тот же тип возвращаемого значения. Создадим два делегата:

```
delegate int Operation(int x, int y);
delegate void GetMessage();
```

Для объявления делегата используется ключевое слово **delegate**, после которого идет возвращаемый тип, название и параметры. Первый делегат ссылается на функцию, которая в качестве параметров принимает два значения типа **int** и возвращает некоторое число. Второй делегат ссылается на метод без параметров, который ничего не возвращает.

Чтобы использовать делегат, нам надо создать его объект с помощью конструктора, в который мы передаем адрес метода, вызываемого делегатом. Чтобы вызвать метод, на который указывает делегат, надо использовать его метод **Invoke**. Кроме того, делегаты могут выполняться в асинхронном режиме, при этом нам не надо создавать второй поток, нам надо лишь вместо метода **Invoke** использовать пару методов **BeginInvoke/EndInvoke**:

```
class Program
{
    delegate void GetMessage(); // 1. Объявляем делегат

    static void Main(string[] args)
    {
        GetMessage del; // 2. Создаем переменную делегата
        if (DateTime.Now.Hour < 12)
```

```
        {
            del = GoodMorning; // 3. Присваиваем этой пере-
менной адрес метода
        }
        else
        {
            del = GoodEvening;
        }
        del.Invoke(); // 4. Вызываем метод
        Console.ReadLine();
    }
    private static void GoodMorning()
    {
        Console.WriteLine("Good Morning");
    }
    private static void GoodEvening()
    {
        Console.WriteLine("Good Evening");
    }
}
```

С помощью свойства **DateTime.Now.Hour** получаем текущий час. И в зависимости от времени в делегат передается адрес определенного метода. Обратите внимание, что методы эти имеют то же возвращаемое значение и тот же набор параметров, в данном случае их отсутствие, что и делегат.

Рассмотрим на примере другого делегата:

```
class Program
{
    delegate int Operation(int x, int y);

    static void Main(string[] args)
    {
        // присваивание адреса метода через контруктор
        Operation del = new Operation(Add); // делегат
указывает на метод Add
        int result = del.Invoke(4,5);
        Console.WriteLine(result);

        del = Multiply; // теперь делегат указывает на
метод Multiply
```

```
            result = del.Invoke(4, 5);
            Console.WriteLine(result);

            Console.Read();
        }
        private static int Add(int x, int y)
        {
            return x+y;
        }
        private static int Multiply (int x, int y)
        {
            return x * y;
        }
}
```

Здесь описан способ присваивания делегату адреса метода через конструктор. И поскольку связанный метод, как и делегат, имеет два параметра, то при вызове делегата в метод **Invoke** мы передаем два параметра. Кроме того, так как метод возвращает значение типа **int**, то мы можем присвоить результат работы метода **Invoke** какой-нибудь переменной.

Метод **Invoke()** при вызове делегата можно опустить и использовать сокращенную форму:

```
del = Multiply; // теперь делегат указывает на метод
Multiply
result = del(4, 5);
```

Таким образом, делегат можно вызывать как обычный метод, передавая ему аргументы.

Так же делегаты могут быть параметрами методов:

```
class Program
{
    delegate void GetMessage();

    static void Main(string[] args)
    {
        if (DateTime.Now.Hour < 12)
        {
            Show_Message(GoodMorning);
        }
        else
        {
```

```
            Show_Message(GoodEvening);
        }
        Console.ReadLine();
    }
    private static void Show_Message(GetMessage _del)
    {
        _del.Invoke();
    }
    private static void GoodMorning()
    {
        Console.WriteLine("Good Morning");
    }
    private static void GoodEvening()
    {
        Console.WriteLine("Good Evening");
    }
}
```

Данные примеры, возможно, не показывают истинной силы делегатов, так как нужные методы в данном случае можно вызвать и напрямую без всяких делегатов. Однако наиболее сильная сторона делегатов состоит в том, что они позволяют создать функционал методов обратного вызова, уведомляя другие объекты о произошедших событиях.

Рассмотрим другой пример. Пусть у нас есть класс, описывающий счет в банке:

```
class Account
{
    int _sum; // Переменная для хранения суммы
    int _percentage; // Переменная для хранения процента

    public Account(int sum, int percentage)
    {
        _sum = sum;
        _percentage = percentage;
    }

    public int CurrentSum
    {
        get { return _sum; }
    }
```

```
    public void Put(int sum)
    {
        _sum += sum;
    }

    public void Withdraw(int sum)
    {
        if (sum <= _sum)
        {
            _sum -= sum;
        }
    }

    public int Percentage
    {
        get { return _percentage; }
    }
}
```

Допустим, в случае вывода денег с помощью метода **Withdraw** нам надо как-то уведомлять об этом самого клиента и, может быть, другие объекты. Для этого создадим делегат **AccountStateHandler**. Чтобы использовать делегат, нам надо создать переменную этого делегата, а затем присвоить ему метод, который будет вызываться делегатом.

Итак, добавим в класс **Account** следующие строки:

```
class Account
{
    // Объявляем делегат
    public delegate void AccountStateHandler(string message);
    // Создаем переменную делегата
    AccountStateHandler del;

    // Регистрируем делегат
    public void RegisterHandler(AccountStateHandler _del)
    {
        del = _del;
    }

    // Далее остальные строки класса Account
```

Здесь фактически проделываются те же шаги, что были выше, и есть практически все, кроме вызова делегата. В данном случае делегат принимает параметр типа **string**. Теперь изменим метод **Withdraw** следующим образом:

```
public void Withdraw(int sum)
{
    if (sum <= _sum)
    {
         _sum -= sum;

        if (del != null)
            del("Сумма " + sum.ToString() + " снята со
счета");
    }
    else
    {
        if (del != null)
            del("Недостаточно денег на счете");
    }
}
```

При снятии денег через метод **Withdraw** сначала проверяем, имеет ли делегат ссылку на какой-либо метод, иначе он имеет значение **null**. И если метод установлен, то вызываем его, передавая соответствующее сообщение в качестве параметра.

Теперь протестируем класс в основной программе:

```
class Program
{
    static void Main(string[] args)
    {
        // создаем банковский счет
        Account account = new Account(200, 6);
        // Добавляем в делегат ссылку на метод Show_Message
        // а сам делегат передается в качестве параметра
метода RegisterHandler
        account.RegisterHandler(new Ac-
count.AccountStateHandler(Show_Message));
        // Два раза подряд пытаемся снять деньги
        account.Withdraw(100);
        account.Withdraw(150);
        Console.ReadLine();
    }
    private static void Show_Message(String message)
    {
        Console.WriteLine(message);
    }
}
```

Запустив программу, получим два разных сообщения:

```
Сумма 100 снята со счета
Недостаточно денег на счете
```

Таким образом, создали механизм обратного вызова для класса **Account**, который срабатывает в случае снятия денег. Поскольку делегат объявлен внутри класса **Account**, то чтобы к нему получить доступ, используется выражение **Account.AccountStateHandler**.

Опять же может возникнуть вопрос: почему бы в коде метода **Withdraw()** не выводить сообщение о снятии денег? Зачем нужно задействовать какой-то делегат?

Дело в том, что не всегда есть доступ к коду классов. Например, часть классов может создаваться и компилироваться одним человеком, который не будет знать, как они будут использоваться. А использовать их будет другой разработчик.

Поэтому необходимо выводить сообщение на консоль. Однако для класса **Account** не важно, как это сообщение выводится. Классу **Account** даже не известно, что вообще будет делаться в результате списания денег. Он просто посылает уведомление об этом через делегат.

Так, через делегата можно выводить сообщение на консоль, создавая консольное приложение. Если создаем графическое приложение **Windows Forms** или **WPF**, то можно выводить сообщение в виде графического окна или записать при списании информацию об этом действии в файл, или отправить уведомление на электронную почту. В общем, любыми способами обработать вызов делегата. И способ обработки не будет зависеть от класса **Account**.

Хотя в примере делегат принимал адрес на один метод, в действительности он может указывать сразу на несколько. Кроме того, при необходимости можно указать ссылки на адреса определенных методов, чтобы они не вызывались при вызове делегата. Итак, изменим в классе **Account** метод **RegisterHandler** и добавим новый метод **UnregisterHandler**, который будет удалять методы из списка методов делегата:

```
// Регистрируем делегат
public void RegisterHandler(AccountStateHandler _del)
{
    Delegate mainDel = System.Delegate.Combine(_del, del);
    del = mainDel as AccountStateHandler;
}
```

```
// Отмена регистрации делегата
public void UnregisterHandler(AccountStateHandler _del)
{
    Delegate mainDel = System.Delegate.Remove(del, _del);
    del = mainDel as AccountStateHandler;
}
```

Метод **Combine** в первом методе объединяет делегаты **_del** и **del** в один, который потом присваивается переменной **del**. Во втором методе метод **Remove** возвращает делегат, из списка вызовов которого удален делегат **_del**. Теперь перейдем к основной программе:

```
class Program
{
    static void Main(string[] args)
    {
        Account account = new Account(200, 6);
        Account.AccountStateHandler colorDelegate = new Ac-
count.AccountStateHandler(Color_Message);

        // Добавляем в делегат ссылку на методы
        account.RegisterHandler(new Ac-
count.AccountStateHandler(Show_Message));
        account.RegisterHandler(colorDelegate);
        // Два раза подряд пытаемся снять деньги
        account.Withdraw(100);
        account.Withdraw(150);

        // Удаляем делегат
        account.UnregisterHandler(colorDelegate);
        account.Withdraw(50);

        Console.ReadLine();
    }
    private static void Show_Message(String message)
    {
        Console.WriteLine(message);
    }
    private static void Color_Message(string message)
    {
        // Устанавливаем красный цвет символов
        Console.ForegroundColor = ConsoleColor.Red;
        Console.WriteLine(message);
        // Сбрасываем настройки цвета
        Console.ResetColor();
    }
}
```

В целях тестирования создадим еще один метод – **Color_Message**, который выводит то же самое сообщение только красным цветом. Для первого делегата создается отдельная переменная. Но большой разницы между передачей обоих в метод **account.RegisterHandler** нет: просто в одном случае мы сразу передаем объект, создаваемый конструктором **account.RegisterHandler(new Account.AccountStateHandler(Show_Message)).**

Во втором случае создаем переменную и ее уже передаем в метод **account.RegisterHandler(colorDelegate)**.

В строке **account.UnregisterHandler(colorDelegate)** этот метод удаляется из списка вызовов делегата, поэтому этот метод больше не будет срабатывать. Консольный вывод будет иметь следующую форму:

```
Сумма 150 снята со счета
Сумма 150 снята со счета
Недостаточно денег на счете
Недостаточно денег на счете
Сумма 50 снята со счета
```

Также можно использовать сокращенную форму добавления и удаления делегатов. Для этого перепишем методы **RegisterHandler** и **UnregisterHandler** следующим образом:

```
// Регистрируем делегат
public void RegisterHandler(AccountStateHandler _del)
{
    del += _del; // добавляем делегат
}
// Отмена регистрации делегата
public void UnregisterHandler(AccountStateHandler _del)
{
    del -= _del; // удаляем делегат
}
```

С помощью делегатов можно создавать механизм обратных вызовов в программе. Однако C# для той же цели предоставляет более удобные и простые конструкции под названием события, которые сигнализируют системе о том, что произошло определенное действие.

События объявляются в классе с помощью ключевого слова **event**, после которого идет название делегата:

```
    // Объявляем делегат
    public delegate void AccountStateHandler(string
message);
    // Событие, возникающее при выводе денег
    public event AccountStateHandler Withdrowed;
```

Связь с делегатом означает, что метод, обрабатывающий данное событие, должен принимать те же параметры и возвращать тот же тип, что и делегат.

Итак, посмотрим на примере. Для этого возьмем класс **Account** и изменим его следующим образом:

```
class Account
{
    // Объявляем делегат
    public delegate void AccountStateHandler(string message);
    // Событие, возникающее при выводе денег
    public event AccountStateHandler Withdrowed;
    // Событие, возникающее при добавление на счет
    public event AccountStateHandler Added;

    int _sum; // Переменная для хранения суммы
    int _percentage; // Переменная для хранения процента

    public Account(int sum, int percentage)
    {
        _sum = sum;
        _percentage = percentage;
    }

    public int CurrentSum
    {
        get { return _sum; }
    }

    public void Put(int sum)
    {
        _sum += sum;
        if (Added != null)
            Added("На счет поступило " + sum);
    }
    public void Withdraw(int sum)
    {
        if (sum <= _sum)
        {
            _sum -= sum;
            if (Withdrowed != null)
                Withdrowed("Сумма " + sum + " снята со
счета");
        }
        else
        {
            if (Withdrowed != null)
```

```
                Withdrowed("Недостаточно денег на счете");
        }
    }

    public int Percentage
    {
        get { return _percentage; }
    }
}
```

Здесь определены два события – **Withdrowed** и **Added**, которые объявлены как экземпляры делегата **AccountStateHandler**. Поэтому для обработки этих событий потребуется метод, принимающий строку в качестве параметра.

Затем в методах **Put** и **Withdraw** вызываем эти события. Перед вызовом проверяем, закреплены ли за ними обработчики:

```
if (Withdrowed != null)
```

Так как эти события представляют делегат **AccountStateHandler**, принимающий в качестве параметра строку, то и при вызове событий передаем в них строку.

Теперь используем события в основной программе:

```
class Program
{
    static void Main(string[] args)
    {
        Account account = new Account(200, 6);
        // Добавляем обработчики события
        account.Added += Show_Message;
        account.Withdrowed += Show_Message;

        account.Withdraw(100);
        // Удаляем обработчик события
        account.Withdrowed -= Show_Message;

        account.Withdraw(50);
        account.Put(150);

        Console.ReadLine();
    }
    private static void Show_Message(string message)
    {
        Console.WriteLine(message);
    }
}
```

Для прикрепления обработчика события к определенному событию используется операция «+=» и соответственно для открепления «– =»: событие += метод_обработчика_события. Опять обращаем внимание, что метод обработчика должен иметь такие же параметры, как и делегат события, и возвращать тот же тип. В итоге получим следующий консольный вывод:

```
Сумма 100 снята со счета
На счет поступило 150
```

Кроме использованного способа прикрепления обработчиков есть и другой с использованием делегата. Но оба способа будут равноценны:

```
account.Added += Show_Message;
account.Added += new Ac-
count.AccountStateHandler(Show_Message);
```

При создании графических приложений с помощью Windows Forms или WPF, можно столкнуться с обработчиками, которые в качестве параметра принимают аргумент типа **EventArgs**. Например, обработчик нажатия кнопки **private void button1_Click(object sender, System.EventArgs e){}.** Параметр «**e»**, будучи объектом класса **EventArgs**, содержит все данные события. Добавим в программу подобный класс, назовем его **AccountEventArgs** и дополним его следующим кодом:

```
class AccountEventArgs
{
    // Сообщение
    public string message;
    // Сумма, на которую изменился счет
    public int sum;

    public AccountEventArgs(string _mes, int _sum)
    {
        message = _mes;
        sum = _sum;
    }
}
```

Данный класс имеет два поля: **message** – для хранения выводимого сообщения и **sum** – для хранения суммы, на которую изменился счет.

Теперь применим класс **AccoutEventArgs**, изменив класс **Account** следующим образом:

```
class Account
{
    // Объявляем делегат
    public delegate void AccountStateHandler(object sender,
AccountEventArgs e);
```

```
    // Событие, возникающее при выводе денег
    public event AccountStateHandler Withdrowed;
    // Событие, возникающее при добавлении на счет
    public event AccountStateHandler Added;

    int _sum; // Переменная для хранения суммы
    int _percentage; // Переменная для хранения процента

    public Account(int sum, int percentage)
    {
        _sum = sum;
        _percentage = percentage;
    }

    public int CurrentSum
    {
        get { return _sum; }
    }

    public void Put(int sum)
    {
        _sum += sum;
        if (Added != null)
            Added(this, new AccountEventArgs("На счет посту-
пило " + sum, sum));
    }
    public void Withdraw(int sum)
    {
        if (sum <= _sum)
        {
            _sum -= sum;
            if (Withdrowed != null)
                Withdrowed(this, new AccountEventArgs("Сумма
" + sum + " снята со счета", sum));
        }
        else
        {
            if (Withdrowed != null)
                Withdrowed(this, new
AccountEventArgs("Недостаточно денег на счете", sum));
        }
    }

    public int Percentage
    {
        get { return _percentage; }
    }
}
```

По сравнению с предыдущей версией класса **Account** здесь изменилось только количество параметров у делегата и соответственно количество параметров при вызове события. Теперь они также принимают объект **AccountEventArgs**, который хранит информацию о событии, получаемую через конструктор.

Теперь изменим основную программу:

```
class Program
{
    static void Main(string[] args)
    {
        Account account = new Account(200, 6);
        // Добавляем обработчики события
        account.Added += Show_Message;
        account.Withdrowed += Show_Message;

        account.Withdraw(100);
        // Удаляем обработчик события
        account.Withdrowed -= Show_Message;

        account.Withdraw(50);
        account.Put(150);

        Console.ReadLine();
    }
    private static void Show_Message(object sender,
AccountEventArgs e)
    {
        Console.WriteLine("Сумма транзакции: {0}", e.sum);
        Console.WriteLine(e.message);
    }
}
```

По сравнению с предыдущим вариантом здесь изменяем только количество параметров и сущность их использования в обработчике **Show_Message**.

С делегатами тесно связано понятие анонимных методов. Анонимные методы представляют сокращенную запись методов. Например, в лабораторной работе №4 было создано событие и обработчик к нему, который выглядел следующим образом:

```
private static void Show_Message(object sender,
AccountEventArgs e)
{
    Console.WriteLine("Сумма транзакции: {0}", e.sum);
    Console.WriteLine(e.message);
}
```

Иногда такие методы нужны для обработки только одного события и больше ценности не представляют, и нигде не используются. Анонимные методы позволяют встроить код там, где он вызывается, например:

```
Account account = new Account(200, 6);
// Добавляем обработчики события
account.Added += delegate(object sender, AccountEventArgs e)
{
    Console.WriteLine("Сумма транзакции: {0}", e.sum);
    Console.WriteLine(e.message);
};
```

Практически это тот же самый метод, только встроенный в код. Встраивание происходит с помощью ключевого слова **delegate**, после которого идет список параметров и далее сам код анонимного метода. Итоговый результат будет такой же, как будто мы подключали обработчик события **Show_Message**.

И важно отметить, что в отличие от блока методов или условных и циклических конструкций, блок анонимных методов должен заканчиваться точкой с запятой после закрывающей фигурной скобки.

Если для анонимного метода не требуется параметров, то он используется без скобок:

```
delegate void GetMessage();
static void Main(string[] args)
{
    GetMessage message = delegate
    {
        Console.WriteLine("анонимный делегат");
    };
    message();

    Console.Read();
}
```

Задания на лабораторную работу

Переопределив с помощью наследования событие, реализуйте обработку ошибок для лабораторной работы №4:

1) StackOverflowException;

2) ArrayTypeMismatchException;

3) DivideByZeroException;

4) IndexOutOfRangeException;

5) InvalidCastException;

6) OutOfMemoryException;

7) OverflowException.

Подготовьте отчет о проделанной работе.

Контрольные вопросы

1. Что понимается под термином «событие»?

2. Являются ли события членами классов?

3. Какое ключевое слово языка C# используется для описания событий?

4. Опишите механизм поддержки событий языка C#.

5. Приведите синтаксис описания события в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

6. Что понимается под термином «широковещательное событие»?

7. На основе какого механизма языка C# строятся широковещательные события?

8. Приведите синтаксис описания широковещательного события в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

9. Что понимается под термином «исключительная ситуация (исключение)»?

10. В чем состоит значение механизма исключений в языке C#?

11. Какие операторы языка C# используются для обработки исключений?

12. Какие операторы языка C# являются важнейшими для обработки исключений?

13. Приведите синтаксис блока **try…catch** в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

14. Приведите пять видов основных системных исключений.

15. Необходимо ли обеспечивать соответствие типов исключения в операторе **catch** типу перехватываемого исключения?

16. Что происходит в случае неудачного перехвата исключения?

17. В каком случае возможно использование оператора **catch** языка C# без параметров?

18. Каким образом осуществляется возврат в программу после обработки исключительной ситуации?

19. Какой оператор языка C# используется для обеспечения возврата в программу после обработки исключения?

20. Приведите синтаксис блока **finally** в составе оператора **try…catch** в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

21. Зависит ли вызов блока **finally** от наличия исключения?

22. Какие способы генерации исключений Вам известны?

23. Что является источником автоматически генерируемых неявных исключений?

24. Каким образом возможно осуществить явную генерацию исключений?

25. Какой оператор языка C# используется для явной генерации исключений?

26. Приведите синтаксис оператора **throw** в общем виде. Проиллюстрируйте его фрагментом программы на языке C#.

27. Каким образом осуществляется повторный перехват исключений в языке C#?

28. Возможно ли создавать специализированные исключения для обработки ошибок в коде пользователя?

29. Какой системный класс является базовым для создания исключений?

30. На основе какого системного класса осуществляется генерация пользовательских исключений?

31. Необходима ли явная реализация классов, наследуемых от системных исключений?

32. Каким образом обеспечивается обращение к свойствам и методам системных исключений?

Приложение

ПРИМЕР ОФОРМЛЕНИЯ ОТЧЕТА О ЛАБОРАТОРНОЙ РАБОТЕ

КГЭУ

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ
Федеральное государственное бюджетное
образовательное учреждение высшего образования
«КАЗАНСКИЙ ГОСУДАРСТВЕННЫЙ
ЭНЕРГЕТИЧЕСКИЙ УНИВЕРСИТЕТ»

Кафедра «Инженерная кибернетика»

О Б Ъ Е К Т Н О - О Р И Е Н Т И Р О В А Н Н О Е
П Р О Г Р А М М И Р О В А Н И Е

Отчет о лабораторной работе №2

Разработка программ для работы со структурами данных

Исполнитель: Садриев Ильмас
Группа: ПОВТ – 1 – 15
Вариант № 7
Дата выполнения 23 января 2018 г.

Дата сдачи _____ **Оценка _____**

Подпись преподавателя _____

Казань
2018

Задача №1

1. Постановка задачи

В текстовом файле хранится база данных отдела кадров предприятия. На предприятии 100 сотрудников. Каждая строка файла содержит запись об одном сотруднике. Формат записи: фамилия – 10 позиций (начинается с первой позиции); год рождения – 6 позиций; оклад – 6 позиций. Составьте программу, которая по заданной фамилии выводит на экран и записывает в выходной файл сведения о сотруднике, подсчитывая средний оклад всех запрошенных сотрудников.

2. Алгоритм решения задачи

Сведения о сотрудниках считать из файла и записать в символьный массив. Организовать цикл запроса сведений о сотруднике: ввод фамилии (с клавиатуры); поиск фамилии сотрудника в массиве; увеличить счетчик количества сотрудников и суммарный оклад; вывести сведения о сотруднике и его среднем окладе или сообщение об их отсутствии.

3. Листинг программы

```
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;
// преобразование DOS to Win
 void DosToWin(char *s1 , char *s2, int  Lname)
 {
   int kod;
   for(int i=0; s1[i] != 0 ; i++)
   {
     kod = (int) s1[i];
     if (( kod>= - 128) && ( kod<= - 81) )
        s2[i] =  (kod+64);
     if (( kod>= - 32) && ( kod<= - 17))
        s2[i] =  (kod+16);
     if (kod== - 16)   s2[i]=   - 88;
     if ( kod== - 15)  s2[i] =  - 72;
    }
 }
int main( )
{
setlocale(0, "");
```

```
const int  Lname  = 10, //фамилия
       Lyear = 6, // год рождения
       Lpay = 6, // оклад
  Lbuf = Lname + Lyear + Lpay;  //  длина буфера
struct Man  // структура
{   // поля структуры
  int birth_year;     // год рождения
  char  name[Lname + 1];  // фамилия
  float  pay;             // оклад
};
const int  Ldbase = 10;  // кол – во сотрудников
Man  dbase[Ldbase];  //определение массива структур
char  buf[Lbuf + 1]; // буфер для ввода строки
                       //  из файла
char  *name = new char [Lname + 1];  // для ввода фамилии
запрашиваемого сотрудника
ifstream fin("TextFile2.txt"); // открытие файла для чтения
ofstream fout("Out.txt");   // создание файла для записи
результатов
if(!fin)
{
  cout << "Ошибка открытия файла" << endl;
  return 1;
}
int i=0;  //  номер текущей строки
// цикл для построчного считывания из файла
while(fin.getline(buf, Lbuf))
{
if(i>=Ldbase)  // проверка на превышение считанного кол –
ва строк размерности массива
{
  cout << "Слишком длинный файл  ";
  return 2;
}
// копирование считанной строки
strncpy(dbase[i].name, buf, Lbuf);
// запись нуль – символа после фамилии
dbase[i].name[Lname]='\0';
// преобразование строки в целое число
dbase[i].birth_year = atoi(&buf[Lname]);
```

```
// &buf[Lname]  -  это адрес начала начала подстроки,
// в которой находится год рождения (после фамилии)
// преобразование строки в вещественное число
// &buf[Lname + Lyear] -  это адрес начала подстроки,
// в которой находится зарплата (после года рождения)
dbase[i].pay = atof(&buf[Lname + Lyear]);
  i++;
}
int Nrecord = i,  // кол - во записей
Nman = 0;    // кол - во просмотренных сотрудников
// сумма окладов сотрудников, для которых выводились
сведения
float  mean_pay = 0;
// бесконечный цикл поиска сотрудника по фамилии
// с принудительным выходом
//(при вводе end вместо фамилии)
while(true)
{
cout << "Введите фамилию или слово end" << endl;
cin >> name;
// преобразование кодировки
DosToWin(name, name, Lname);
// проверка ввода "end"
if(strcmp(name, "end") == 0 ) break;
// переменная - флаг "фамилия не найдена"
bool not_found = true;
// цикл просмотра массива структур
for(i=0; i< Nrecord; i++)
{// проверка совпадения фамилии
if((strstr(dbase[i].name, name)))
// есть - ли после фамилии пробел?
if(dbase[i].name[strlen(name)] ==' ')
{
 cout << setw(10) << name << ": год рожд." <<
dbase[i].birth_year << " оклад " << dbase[i].pay  << endl;
fout << setw(10)  << name << ": год рожд." <<
dbase[i].birth_year << " оклад " << dbase[i].pay  << endl;
Nman++;  // кол - во просмотренных сотрудников
// сумма окладов сотрудников, для которых выводились
сведения
```

```
mean_pay += dbase[i].pay;
not_found = false;  // фамилия найдена
}
}
if(not_found) cout << " Такого сотрудника нет "  << endl;
}
// при завершении работы программы
// выводится средний оклад
if(Nman > 0)
{
cout << "Средний оклад "  << mean_pay / Nman << endl; //
для которых выводились сведения
fout << "Средний оклад "  << mean_pay / Nman << endl; //
для которых выводились сведения
}
fout.close();
return 0;
}
```

4. Исходные данные

Исходные данные содержатся в файле TextFile2.txt:

```
Ivanov    1993   8500
Петров    1994  12100
Sidorov   1993  10800
Петров    1995  15555
```

5. Результаты работы программы

Результаты работы с программой на экране:

```
Введите фамилию или слово end
Ivanov
    Ivanov: год рожд.1993 оклад 8500
Введите фамилию или слово end
Sidorov
   Sidorov: год рожд.1993 оклад 10800
Введите фамилию или слово end
Петров
    Петров: год рожд.1994 оклад 12100
    Петров: год рожд.1995 оклад 15555
Введите фамилию или слово end
end
Средний оклад 11738.8
```

Содержимое выходного файла:

```
    Ivanov: год рожд.1993 оклад 8500
    Sidorov: год рожд.1993 оклад 10800
    Петров: год рожд.1994 оклад 12100
    Петров: год рожд.1995 оклад 15555
Средний оклад 11738.8
```

Задача №2

1. Постановка задачи

В текстовом файле хранится база данных отдела кадров предприятия. На предприятии 100 сотрудников. Каждая строка файла содержит запись об одном сотруднике. Формат записи: фамилия – 10 позиций (начинается с первой позиции); год рождения – 6 позиций; оклад – 6 позиций. Составьте программу, которая упорядочивает записи по году рождения сотрудников. Результат запишите в выходной файл.

2. Алгоритм решения задачи

За основу взять программу поиска в массиве структур. Изменить ее так, чтобы она вместо поиска упорядочивала массив, а затем записывала его в новый файл.

Для сортировки применить метод выбора: из массива выбрать наименьший элемент и поменять местами с первым элементом, затем рассматривать элементы, начиная со второго, наименьший из них поменять местами со вторым элементом, и т.д.

При последнем проходе цикла при необходимости поменять местами предпоследний и последний элементы массива, так как для упорядочения требуется количество просмотров на единицу меньше, чем количество элементов в массиве.

3. Листинг программы

```
#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;
int main( )
{
setlocale(0, "");
const int  Lname  = 10, //фамилия инициалы
       Lyear = 6, // год рождения
       Lpay = 6, // оклад
Lbuf = Lname + Lyear + Lpay;  //  длина буфера
struct Man  // структура
{    // поля структуры
  int birth_year;     // год рождения
  char  name[Lname + 1];  // фамилия
  float  pay;             // оклад
};
```

```
const int  Ldbase = 10;
Man  dbase[Ldbase];  //определение массива структур
// буфер для ввода строки из файла
char  buf[Lbuf + 1];
// для ввода фамилии запрашиваемого сотрудника
char  name[Lname + 1];
// открытие файла для чтения
ifstream fin("TextFile2.txt");
if(!fin)
{
  cout << "Ошибка открытия файла" << endl;
  return 1;
}
int i=0;  // номер записи о сотруднике
// цикл для построчного считывания из файла
while(fin.getline(buf, Lbuf))
{ // копирование считанной строки
     strncpy(dbase[i].name, buf, Lbuf);
     // запись нуль – символа после фамилии
     dbase[i].name[Lname]='\0';
     // преобразование строки в целое число
  dbase[i].birth_year = atoi(&buf[Lname]);
  // &buf[Lname]  -  это адрес начала начала
     // подстроки, в которой находится
     // год рождения (после фамилии)
// преобразование строки в вещественное число
dbase[i].pay = atof(&buf[Lname + Lyear]);
// &buf[Lname + Lyear] -  это адрес начала подстроки,
//  в которой находится зарплата
// (после года рождения)
i++;
if(i>=Ldbase)
{// проверка на превышение считанного кол – ва строк
размерности массива
cout << "Слишком длинный файл  ";
return 2;
}
}
int Nrecord = i;  // кол – во записей
fin.close();
```

```
ofstream fout("dbase.txt");
for(i=0; i<Nrecord – 1; i++)
{
//примем за наименьший первый
//из рассматриваемых элементов
int imin = i;
// поиск номера минимального элемента
// из неупорядоченных
for(int j=i+1; j<Nrecord; j++)
if(dbase[j].birth_year < dbase[imin].birth_year)
imin = j;
// обмен двух элементов массива структур
Man a = dbase[i]; // для временного хранения
dbase[i]=dbase[imin];
dbase[imin]=a;
}
for(i=0; i<Nrecord; i++)
{ cout <<dbase[i].name << dbase[i].birth_year << " " <<
dbase[i].pay << endl;
fout << dbase[i].name << dbase[i].birth_year << " " <<
dbase[i].pay << endl;
}
fout.close( );
cout << "Сортировка завершена "  << endl; // для которых
выводились сведения
return 0;
}
```

4. Исходные данные.

Исходные данные содержатся в файле TextFile2.txt:

```
ivanov    1994   8500
петров    1995  12100
sidorov   1993  10800
```

5. Результаты работы программы.

Результаты работы с программой на экране:

```
sidorov   1993 10800
ivanov    1994 8500
петров    1995 12100
Сортировка завершена
```

Содержимое выходного файла:

```
sidorov   1993 10800
ivanov    1994 8500
петров    1995 12100
```

5. Выводы.

В результате выполнения лабораторной работы убедились, что при программировании задач структуры удобно использовать логическое объединение связанных между собой данных, что позволяет оперировать с ними как с единым объектом. В структуры, в противоположность массиву, можно объединять данные различных типов.

Кроме того, в результате выполнения лабораторной работы мы приобрели навыки программирования задач с использованием структур данных, а также приобрели навыки по вводу программ и исходных данных, отладке, компиляции и контролю правильности решения задач на компьютере.

6. Подпись исполнителя

СОДЕРЖАНИЕ

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Анисимов, А. Е. Сборник заданий по основам программирования: учебное пособие / А. Е. Анисимов, В. В. Пупышев. – М.: Изд-во «БИНОМ. Лаборатория знаний», 2006. – 348 с.

2. Головин, И. Г. Языки и методы программирования: учебник / И. Г. Головин, И. А. Волкова. – М.: Издательский центр «Академия», 2012. – 304 с. – (Сер. «Бакалавриат»).

3. Дейтел, П. Как программировать на Visual C# [Электронный ресурс] / П. Дейтел, Х. Дейтел. – 5-е изд. – СПб.: Питер, 2014. – 864 с. – (Сер. «Библиотека программиста»). – Режим доступа: http://ibooks.ru/reading.php?productid=341183. – Загл. с экрана.

4. Задачи по программированию: задачник / под ред. С. М. Окулова. – М.: Издв-во «БИНОМ. Лаборатория знаний», 2006. – 820 с.

5. Одинцов, И. О. Профессиональное программирование. Системный подход [Электронный ресурс] / И. О. Одинцов. – 2-е изд. перер. и доп. – СПб.: БХВ-Петербург, 2014. – 624 с. – Режим доступа: http://ibooks.ru/reading.php? productid=18535. – Загл. с экрана.

6. Орлов, С. А. Теория и практика языков программирования [Электронный ресурс]: учебник для вузов / С. А. Орлов. – СПб.: Питер, 2013. – 688 с. – Режим доступа: http://ibooks.ru/reading.php? productid=26402. – Загл. с экрана.

7. Служба разработчиков Майкрософт [Электронный ресурс]. – Режим доступа: http://msdn.microsoft.com/ru-ru. – Загл. с экрана. – (Дата обращения: 12.10.2017).

8. Форум программистов и сисадминов Киберфорум [Электронный ресурс]. – Режим доступа: http://CyberForum.ru. – Загл. с экрана. – (Дата обращения: 27.09.2017).

9. Форум, учебники, исходники для программистов [Электронный ресурс]. – Режим доступа: http://olocoder.ru. – Загл. с экрана. – (Дата обращения: 27.11.2017).

10. Хорев, П. Б. Объектно-ориентированное программирование: учебное пособие / П. Б. Хорев. – 4-е изд., стер. – М.: Издательский центр «Академия», 2012. – 448 с. – (Сер. «Бакалавриат»).

11. CodeNet – все для программиста [Электронный ресурс]. – Режим доступа: http://codenet.ru. – Загл. с экрана. – (Дата обращения: 21.09.2017).

Учебное издание

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Практикум

Составитель: Халидов Али Анварович

Кафедра инженерной кибернетики КГЭУ

Редактор издательского отдела И.В. Краснова
Компьютерная верстка И.В. Краснова

Подписано в печать 14.03.18.
Формат 60×84/16. Бумага ВХИ. Гарнитура «Times». Вид печати РОМ.
Усл. печ. л. 4,82. Уч.-изд. л. 2,70. Тираж 500 экз. Заказ № 186/эл.

Редакционно-издательский отдел КГЭУ
420066, г. Казань, ул. Красносельская, 51